**Step Four: Pre-processing and Training Data Development**

Data pre-processing refers to the process of removing things like out-of-value ranges and impossible combinations from your dataset. This is an important step because analyzing data that hasn’t been properly screened can lead to misleading or incorrect results. This step also includes the process of splitting the dataset into testing and training subsets.



[Write](https://medium.com/m/signin?operation=register&redirect=https%3A%2F%2Fmedium.com%2Fnew-story&source=---two_column_layout_nav-----------------------new_post_sidenav-----------)

[Sign up](https://medium.com/m/signin?operation=register&redirect=https%3A%2F%2Ftowardsdatascience.com%2Fpre-processing-and-training-data-d16cc12dfbac&source=post_page---two_column_layout_nav-----------------------global_nav-----------)

[Sign In](https://medium.com/m/signin?operation=login&redirect=https%3A%2F%2Ftowardsdatascience.com%2Fpre-processing-and-training-data-d16cc12dfbac&source=post_page---two_column_layout_nav-----------------------global_nav-----------)

[[Towards Data Science](https://towardsdatascience.com/?source=post_page-----d16cc12dfbac--------------------------------)](https://towardsdatascience.com/?source=post_page-----d16cc12dfbac--------------------------------)

Published in

[Towards Data Science](https://towardsdatascience.com/?source=post_page-----d16cc12dfbac--------------------------------)

[[](https://guymaskall.medium.com/?source=post_page-----d16cc12dfbac--------------------------------)](https://guymaskall.medium.com/?source=post_page-----d16cc12dfbac--------------------------------)

[Guy Maskall](https://guymaskall.medium.com/?source=post_page-----d16cc12dfbac--------------------------------)

Jul 15, 2020

·

5 min read

·

# Pre-processing and training data

## Transforming your data and training a model for best performance is a holistic process.

A picture containing electronics, circuit

Description automatically generated

Image by [Gerd Altmann](https://pixabay.com/users/geralt-9301/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=3706562) from [Pixabay](https://pixabay.com/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=3706562)

# Pre-processing

So, what is pre-processing? Common examples include

* imputing missing values
* transforming the values (linearly or nonlinearly)
* Encoding categorical variables

## Imputing missing values

There are often holes in your data. Someone didn’t respond to a survey question. A sensor glitched. The data got lost. There are all kinds of missing value and causes thereof. Missing values can be denoted by a special ‘value’ of NaN (in Python, from NumPy) or NA (in R). But missingness isn’t always denoted by these special markers. Negative values can denote missing data, for example -999 as someone’s age. This can depend on the originating system or software as well as how the data was handled up to the point you received it. An empty string may or may not be classed as “missing” in your context.

An important question that you should have tackled by now is whether there seems to be any pattern in the missing data. You did explore this in your preceding EDA stage, right? Were those river level sensor readings missing because the station was flooded? Or are the missing values scattered randomly throughout the data, suggesting some more innocuous reason such as intermittent radio interference perhaps? If the former, then I hope you’ve added a new feature capturing the fact that the values were missing. Regardless, most machine learning algorithms don’t like missing values so you need to take steps such as, well frankly, guessing what the value might have been. This is imputing.

There are any number of schemes for imputing missing values. You could use the mean of the remaining values. Or the median. Or take the last known value. Yes, picking the best strategy can involve trial and error.

## Transforming the values

There are a few reasons to want to transform the values of your features. You may have highly skewed distributions, and a nonlinear transformation can tend to normalize these, which can help many algorithms. You might have a collection of features measured on very different scales, for example land elevation in feet and area in acres. Or people’s weight in pounds and body temperature in degrees Fahrenheit, vs weight in kilograms and body temperature in degrees Celsius. Indeed, these units aren’t so wildly different, but this leads us to another reason to want to transform feature values; algorithms that work off distances between points, and algorithms that regularize (penalize) their coefficients can produce very different results arbitrarily because of such differences in unit. Scaling features puts them all on an equal footing.

## Encoding categorical variables

Algorithms like numbers. They’re much less keen on strings (text). Much of data science is, arguably, the quest to get your data into an appropriate numerical representation that a machine learning algorithm can get to work on it. Rather than have a column named ‘Animal’ with the value of ‘Cat’ or ‘Dog’, you’d probably create a column ‘Cat’ with the value 0 or 1 denoting ‘not a cat’ or ‘is a cat’ and a column ‘Dog’ with value 0 or 1 denoting ‘not a dog’ or ‘is a dog’. Even this relatively simple scheme can have some subtleties such as the need to drop one of those columns for models such as linear regression that include an intercept. This is to avoid the [dummy variable trap](https://www.algosome.com/articles/dummy-variable-trap-regression.html).

# Train-test split

If there are so many possible decisions to make before even training a machine learning model, how do we decide which to use? We could make our pre-processing, and our model, ever more and more complex to fit more and more arbitrary structure in our data. How do we know our pre-processing decisions were the right ones and we weren’t just fitting to noise in our data? This is known as overfitting. The cornerstone of assessing whether a model is overfitting to the data it knows about is to not give it all the data when training it, and hold some back to then see how it performs.

A cunning variation on this is to further partition the training data into multiple splits and train on all but one, and test on the remaining one. By rotating the partition so held back, we end up with a number of estimates of performance on a validation set (where the validation set is different each time). This is cross-validation and is particularly useful in performing model selection. Of course, we can still keep a set of data held back from all of this, to ensure we have some data that our model has never seen. After multiple cross-validation rounds where multiple models were trialled, such a final test set is a useful final check of expected future performance.

# Metrics

But how do we measure performance? There are a number of possible measures, or metrics, and the most appropriate one varies depending on the use case. If we’re performing classification, for example “cat” or “dog”, then a metric that captures the number of times we got the right answer vs the number of times we got it wrong is appropriate. But if we’re doing regression, where we’re trying to predict a continuous variable such as someone’s salary, or a house price, then a measure that reflects how close we tended to be, on average to the true value is best. A common metric here is the mean absolute error, which, as the name suggests, is the average of the magnitudes of the errors.

# Pulling it all together

Now we actually have all the ingredients we need. For our specific type of machine learning problem, we can pick a metric. Having a metric allows us to measure our degree of success (or failure). Having a train-test split gives us a means to apply that metric to a partition of data the algorithm hasn’t been privy to. These together allow us to trial many different types of model and pre-processing steps and assess which one performed the best.

Hey Presto! If you’ve trialled a variety of pre-processing steps and algorithms, and applied an appropriate metric and assessed your entire process against a test set (e.g. via cross-validation), you’ve got a trained model. Specifically, you’ve established the combination of pre-processing steps and the nature of the algorithm.

# About this article

This is the fifth article of a linked series written to provide a straightforward introduction to getting started with the data science process. You can find the introduction [here](https://medium.com/@guymaskall/the-data-science-method-dsm-35200eb4984), the previous article [here](https://medium.com/@guymaskall/exploratory-data-analysis-86eb12060eac), and the next article in the series [here](https://medium.com/@guymaskall/modelling-e096495d8d70).

[Data Science](https://medium.com/tag/data-science?source=post_page-----d16cc12dfbac---------------data_science-----------------)



[Write](https://medium.com/m/signin?operation=register&redirect=https%3A%2F%2Fmedium.com%2Fnew-story&source=---two_column_layout_nav-----------------------new_post_sidenav-----------)

[Sign In](https://medium.com/m/signin?operation=login&redirect=https%3A%2F%2Fmedium.com%2Fgreyatom%2Fwhy-how-and-when-to-scale-your-features-4b30ab09db5e&source=post_page---two_column_layout_nav-----------------------global_nav-----------)

[[GreyAtom](https://medium.com/greyatom?source=post_page-----4b30ab09db5e--------------------------------)](https://medium.com/greyatom?source=post_page-----4b30ab09db5e--------------------------------)

Published in

[GreyAtom](https://medium.com/greyatom?source=post_page-----4b30ab09db5e--------------------------------)

[[](https://medium.com/@sudharsanasai?source=post_page-----4b30ab09db5e--------------------------------)](https://medium.com/@sudharsanasai?source=post_page-----4b30ab09db5e--------------------------------)

[Sudharsan Asaithambi](https://medium.com/@sudharsanasai?source=post_page-----4b30ab09db5e--------------------------------)

Dec 3, 2017

·

3 min read

·

# Why, How and When to Scale your Features

Feature scaling can vary your results a lot while using certain algorithms and have a minimal or no effect in others. To understand this, let’s look why features need to be scaled, varieties of scaling methods and when we should scale our features.

# Why Scaling

Most of the times, your dataset will contain features highly varying in magnitudes, units and range. But since, most of the machine learning algorithms use Eucledian distance between two data points in their computations, this is a problem.

If left alone, these algorithms only take in the magnitude of features neglecting the units. The results would vary greatly between different units, 5kg and 5000gms. The features with high magnitudes will weigh in a lot more in the distance calculations than features with low magnitudes.

A picture containing text, ground, person, person

Description automatically generated

Tiny Features vs Mega Features

To supress this effect, we need to bring all features to the same level of magnitudes. This can be acheived by scaling.

# **How to Scale Features**

There are four common methods to perform Feature Scaling.

1. **Standardisation:**

Standardisation replaces the values by their Z scores.

A picture containing text, watch, clock, gauge

Description automatically generated

This redistributes the features with their mean **μ = 0** and standard deviation **σ =1** . sklearn.preprocessing.scale helps us implementing standardisation in python.

**2. Mean Normalisation:**

Text

Description automatically generated with medium confidence

This distribution will have values between **-1 and 1**with **μ=0**.

**Standardisation** and **Mean Normalization** can be used for algorithms that assumes zero centric data like **Principal Component Analysis(PCA).**

**3. Min-Max Scaling:**

Text

Description automatically generated with medium confidence

This scaling brings the value between 0 and 1.

**4. Unit Vector:**

Diagram

Description automatically generated

Scaling is done considering the whole feature vecture to be of unit length.

**Min-Max Scaling** and **Unit Vector** techniques produces values of range [0,1]. When dealing with features with hard boundaries this is quite useful. For example, when dealing with image data, the colors can range from only 0 to 255.

# **When to Scale**

Rule of thumb I follow here is any algorithm that computes distance or assumes normality, **scale your features!!!**

Some examples of algorithms where feature scaling matters are:

* **k-nearest neighbors** with an Euclidean distance measure is sensitive to magnitudes and hence should be scaled for all features to weigh in equally.
* Scaling is critical, while performing **Principal Component Analysis(PCA)**. PCA tries to get the features with maximum variance and the variance is high for high magnitude features. This skews the PCA towards high magnitude features.
* We can speed up **gradient descent** by scaling. This is because θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven.
* **Tree based models** are not distance based models and can handle varying ranges of features. Hence, Scaling is not required while modelling trees.
* Algorithms like **Linear Discriminant Analysis(LDA), Naive Bayes** are by design equipped to handle this and gives weights to the features accordingly. Performing a features scaling in these algorithms may not have much effect.

Hope you understood the **why**, **how** and **when** of feature scaling.

I would love to hear more about your best practices and rules of thumb. Please **comment** below to enlighten us all with the same.

When working with a dataset, may need to scale your data to achieve clear, accurate results. This process includes adjusting all the features within your dataset so that they have the same magnitude. This resource provides an overview of the four ways you can achieve scale standardization and explains when you should scale your data.

3

**Guided Capstone - Step Four**

Saved

4 - 8 Hours

110 Points

In this step of the guided capstone, you’ll work through pre-processing and training data development! This is the last step needed before you can model your data and is vitally important for good outcomes.

**Steps:**

1. Open the notebook 04\_preprocessing\_and\_training.ipynb, which you’ll find under the folder “Notebooks”
2. Complete the notebook
3. Push the updated notebook to GitHub
4. Submit a link to the notebook on GitHub. Specifically link the location of your notebook, and not just the general link to your GitHub repository.

You can find the rubric [here](https://www.springboard.com/archeio/download/c0c2d0bd209b41619b4004953d84245d/).

Guided Capstone Grading Rubric  
Guided Capstone Step 4: Pre-processing and model development  
Learning Objective  
● Gain a conceptual understanding of important topics such as train/test splits and  
metrics.  
● Understand the importance of gaining an initial baseline performance from a  
non-ML approach (e.g. simply assuming/guessing the mean).  
● Get an understanding of how to vary components in the pipeline easily.  
● Learn how to explore and assess different model hyperparameters via  
cross-validation (as opposed to on a test set) and learn about associated  
functions.  
● Learn how to serialize (pickle) and save a (model) object to file.  
Criteria Meets Expectations  
Completion ❏ Every step in the Jupyter notebook is completed and  
functions correctly.  
Process and  
understanding  
❏ The submission shows that students can perform a  
workflow from imputing missing values, scaling data in the  
context of training, to assessing model performance.  
❏ The submission demonstrates that students can  
implement this workflow as a single pipeline object that  
can be trained and assessed just as a simple model.  
❏ The submission demonstrates that students can compare  
model performances and select the best model.  
Presentation ❏ The appropriate GitHub repository is created and the  
completed notebook pushed to it.  
❏ A link to the specific step 4 notebook is submitted.

[Jupyter Notebook](http://localhost:8890/tree?token=34101dfefd012e8bd0b705731d01b2a4049a6063cf628fdf)

04\_preprocessing\_and\_training Last Checkpoint: an hour ago (autosaved)

Python 3 (ipykernel)

* [File](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [Edit](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [View](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [Insert](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [Cell](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [Kernel](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [Widgets](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [Help](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)

# 4 Pre-Processing and Training Data

## 4.1 Contents

* [4 Pre-Processing and Training Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4_Pre-Processing_and_Training_Data)
  + [4.1 Contents](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.1_Contents)
  + [4.2 Introduction](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.2_Introduction)
  + [4.3 Imports](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.3_Imports)
  + [4.4 Load Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.4_Load_Data)
  + [4.5 Extract Big Mountain Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.5_Extract_Big_Mountain_Data)
  + [4.6 Train/Test Split](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.6_Train/Test_Split)
  + [4.7 Initial Not-Even-A-Model](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7_Initial_Not-Even-A-Model)
    - [4.7.1 Metrics](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.1_Metrics)
      * [4.7.1.1 R-squared, or coefficient of determination](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.1.1_R-squared,_or_coefficient_of_determination)
      * [4.7.1.2 Mean Absolute Error](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.1.2_Mean_Absolute_Error)
      * [4.7.1.3 Mean Squared Error](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.1.3_Mean_Squared_Error)
    - [4.7.2 sklearn metrics](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.2_sklearn_metrics)
      * [4.7.2.0.1 R-squared](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.2.0.1_R-squared)
      * [4.7.2.0.2 Mean absolute error](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.2.0.2_Mean_absolute_error)
      * [4.7.2.0.3 Mean squared error](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.2.0.3_Mean_squared_error)
    - [4.7.3 Note On Calculating Metrics](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.3_Note_On_Calculating_Metrics)
  + [4.8 Initial Models](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8_Initial_Models)
    - [4.8.1 Imputing missing feature (predictor) values](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1_Imputing_missing_feature_(predictor)_values)
      * [4.8.1.1 Impute missing values with median](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1_Impute_missing_values_with_median)
        + [4.8.1.1.1 Learn the values to impute from the train set](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1.1_Learn_the_values_to_impute_from_the_train_set)
        + [4.8.1.1.2 Apply the imputation to both train and test splits](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1.2_Apply_the_imputation_to_both_train_and_test_splits)
        + [4.8.1.1.3 Scale the data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1.3_Scale_the_data)
        + [4.8.1.1.4 Train the model on the train split](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1.4_Train_the_model_on_the_train_split)
        + [4.8.1.1.5 Make predictions using the model on both train and test splits](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1.5_Make_predictions_using_the_model_on_both_train_and_test_splits)
        + [4.8.1.1.6 Assess model performance](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1.6_Assess_model_performance)
      * [4.8.1.2 Impute missing values with the mean](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2_Impute_missing_values_with_the_mean)
        + [4.8.1.2.1 Learn the values to impute from the train set](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2.1_Learn_the_values_to_impute_from_the_train_set)
        + [4.8.1.2.2 Apply the imputation to both train and test splits](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2.2_Apply_the_imputation_to_both_train_and_test_splits)
        + [4.8.1.2.3 Scale the data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2.3_Scale_the_data)
        + [4.8.1.2.4 Train the model on the train split](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2.4_Train_the_model_on_the_train_split)
        + [4.8.1.2.5 Make predictions using the model on both train and test splits](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2.5_Make_predictions_using_the_model_on_both_train_and_test_splits)
        + [4.8.1.2.6 Assess model performance](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2.6_Assess_model_performance)
    - [4.8.2 Pipelines](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.2_Pipelines)
      * [4.8.2.1 Define the pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.2.1_Define_the_pipeline)
      * [4.8.2.2 Fit the pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.2.2_Fit_the_pipeline)
      * [4.8.2.3 Make predictions on the train and test sets](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.2.3_Make_predictions_on_the_train_and_test_sets)
      * [4.8.2.4 Assess performance](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.2.4_Assess_performance)
  + [4.9 Refining The Linear Model](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9_Refining_The_Linear_Model)
    - [4.9.1 Define the pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.1_Define_the_pipeline)
    - [4.9.2 Fit the pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.2_Fit_the_pipeline)
    - [4.9.3 Assess performance on the train and test set](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.3_Assess_performance_on_the_train_and_test_set)
    - [4.9.4 Define a new pipeline to select a different number of features](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.4_Define_a_new_pipeline_to_select_a_different_number_of_features)
    - [4.9.5 Fit the pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.5_Fit_the_pipeline)
    - [4.9.6 Assess performance on train and test data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.6_Assess_performance_on_train_and_test_data)
    - [4.9.7 Assessing performance using cross-validation](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.7_Assessing_performance_using_cross-validation)
    - [4.9.8 Hyperparameter search using GridSearchCV](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.8_Hyperparameter_search_using_GridSearchCV)
  + [4.10 Random Forest Model](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.10_Random_Forest_Model)
    - [4.10.1 Define the pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.10.1_Define_the_pipeline)
    - [4.10.2 Fit and assess performance using cross-validation](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.10.2_Fit_and_assess_performance_using_cross-validation)
    - [4.10.3 Hyperparameter search using GridSearchCV](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.10.3_Hyperparameter_search_using_GridSearchCV)
  + [4.11 Final Model Selection](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.11_Final_Model_Selection)
    - [4.11.1 Linear regression model performance](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.11.1_Linear_regression_model_performance)
    - [4.11.2 Random forest regression model performance](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.11.2_Random_forest_regression_model_performance)
    - [4.11.3 Conclusion](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.11.3_Conclusion)
  + [4.12 Data quantity assessment](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.12_Data_quantity_assessment)
  + [4.13 Save best model object from pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.13_Save_best_model_object_from_pipeline)
  + [4.14 Summary](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.14_Summary)

## 4.2 Introduction

In preceding notebooks, performed preliminary assessments of data quality and refined the question to be answered. You found a small number of data values that gave clear choices about whether to replace values or drop a whole row. You determined that predicting the adult weekend ticket price was your primary aim. You threw away records with missing price data, but not before making the most of the other available data to look for any patterns between the states. You didn't see any and decided to treat all states equally; the state label didn't seem to be particularly useful.

In this notebook you'll start to build machine learning models. Before even starting with learning a machine learning model, however, start by considering how useful the mean value is as a predictor. This is more than just a pedagogical device. You never want to go to stakeholders with a machine learning model only to have the CEO point out that it performs worse than just guessing the average! Your first model is a baseline performance comparitor for any subsequent model. You then build up the process of efficiently and robustly creating and assessing models against it. The development we lay out may be little slower than in the real world, but this step of the capstone is definitely more than just instructional. It is good practice to build up an understanding that the machine learning pipelines you build work as expected. You can validate steps with your own functions for checking expected equivalence between, say, pandas and sklearn implementations.

## 4.3 Imports



import pandas as pd

import numpy as np

import os

import pickle

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn import \_\_version\_\_ as sklearn\_version

from sklearn.decomposition import PCA

from sklearn.preprocessing import scale

from sklearn.model\_selection import train\_test\_split, cross\_validate, GridSearchCV, learning\_curve

from sklearn.preprocessing import StandardScaler, MinMaxScaler

from sklearn.dummy import DummyRegressor

from sklearn.linear\_model import LinearRegression

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import r2\_score, mean\_squared\_error, mean\_absolute\_error

from sklearn.pipeline import make\_pipeline

from sklearn.impute import SimpleImputer

from sklearn.feature\_selection import SelectKBest, f\_regression

import datetime

​

from library.sb\_utils import save\_file

## 4.4 Load Data



ski\_data = pd.read\_csv('../data/ski\_data\_step3\_features.csv')

ski\_data.head().T

|  | **0** | **1** | **2** | **3** | **4** |
| --- | --- | --- | --- | --- | --- |
| **Name** | Alyeska Resort | Eaglecrest Ski Area | Hilltop Ski Area | Arizona Snowbowl | Sunrise Park Resort |
| **Region** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **state** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **summit\_elev** | 3939 | 2600 | 2090 | 11500 | 11100 |
| **vertical\_drop** | 2500 | 1540 | 294 | 2300 | 1800 |
| **base\_elev** | 250 | 1200 | 1796 | 9200 | 9200 |
| **trams** | 1 | 0 | 0 | 0 | 0 |
| **fastSixes** | 0 | 0 | 0 | 1 | 0 |
| **fastQuads** | 2 | 0 | 0 | 0 | 1 |
| **quad** | 2 | 0 | 0 | 2 | 2 |
| **triple** | 0 | 0 | 1 | 2 | 3 |
| **double** | 0 | 4 | 0 | 1 | 1 |
| **surface** | 2 | 0 | 2 | 2 | 0 |
| **total\_chairs** | 7 | 4 | 3 | 8 | 7 |
| **Runs** | 76 | 36 | 13 | 55 | 65 |
| **TerrainParks** | 2 | 1 | 1 | 4 | 2 |
| **LongestRun\_mi** | 1 | 2 | 1 | 2 | 1.2 |
| **SkiableTerrain\_ac** | 1610 | 640 | 30 | 777 | 800 |
| **Snow Making\_ac** | 113 | 60 | 30 | 104 | 80 |
| **daysOpenLastYear** | 150 | 45 | 150 | 122 | 115 |
| **yearsOpen** | 60 | 44 | 36 | 81 | 49 |
| **averageSnowfall** | 669 | 350 | 69 | 260 | 250 |
| **AdultWeekend** | 85 | 53 | 34 | 89 | 78 |
| **projectedDaysOpen** | 150 | 90 | 152 | 122 | 104 |
| **NightSkiing\_ac** | 550 | NaN | 30 | NaN | 80 |
| **resorts\_per\_state** | 3 | 3 | 3 | 2 | 2 |
| **resorts\_per\_100kcapita** | 0.410091 | 0.410091 | 0.410091 | 0.0274774 | 0.0274774 |
| **resorts\_per\_100ksq\_mile** | 0.450867 | 0.450867 | 0.450867 | 1.75454 | 1.75454 |
| **resort\_skiable\_area\_ac\_state\_ratio** | 0.70614 | 0.280702 | 0.0131579 | 0.492708 | 0.507292 |
| **resort\_days\_open\_state\_ratio** | 0.434783 | 0.130435 | 0.434783 | 0.514768 | 0.485232 |
| **resort\_terrain\_park\_state\_ratio** | 0.5 | 0.25 | 0.25 | 0.666667 | 0.333333 |
| **resort\_night\_skiing\_state\_ratio** | 0.948276 | NaN | 0.0517241 | NaN | 1 |
| **total\_chairs\_runs\_ratio** | 0.0921053 | 0.111111 | 0.230769 | 0.145455 | 0.107692 |
| **total\_chairs\_skiable\_ratio** | 0.00434783 | 0.00625 | 0.1 | 0.010296 | 0.00875 |
| **fastQuads\_runs\_ratio** | 0.0263158 | 0 | 0 | 0 | 0.0153846 |
| **fastQuads\_skiable\_ratio** | 0.00124224 | 0 | 0 | 0 | 0.00125 |

## 4.5 Extract Big Mountain Data

Big Mountain is your resort. Separate it from the rest of the data to use later.



big\_mountain = ski\_data[ski\_data.Name == 'Big Mountain Resort']



big\_mountain.T

|  | **124** |
| --- | --- |
| **Name** | Big Mountain Resort |
| **Region** | Montana |
| **state** | Montana |
| **summit\_elev** | 6817 |
| **vertical\_drop** | 2353 |
| **base\_elev** | 4464 |
| **trams** | 0 |
| **fastSixes** | 0 |
| **fastQuads** | 3 |
| **quad** | 2 |
| **triple** | 6 |
| **double** | 0 |
| **surface** | 3 |
| **total\_chairs** | 14 |
| **Runs** | 105 |
| **TerrainParks** | 4 |
| **LongestRun\_mi** | 3.3 |
| **SkiableTerrain\_ac** | 3000 |
| **Snow Making\_ac** | 600 |
| **daysOpenLastYear** | 123 |
| **yearsOpen** | 72 |
| **averageSnowfall** | 333 |
| **AdultWeekend** | 81 |
| **projectedDaysOpen** | 123 |
| **NightSkiing\_ac** | 600 |
| **resorts\_per\_state** | 12 |
| **resorts\_per\_100kcapita** | 1.12278 |
| **resorts\_per\_100ksq\_mile** | 8.16104 |
| **resort\_skiable\_area\_ac\_state\_ratio** | 0.140121 |
| **resort\_days\_open\_state\_ratio** | 0.129338 |
| **resort\_terrain\_park\_state\_ratio** | 0.148148 |
| **resort\_night\_skiing\_state\_ratio** | 0.84507 |
| **total\_chairs\_runs\_ratio** | 0.133333 |
| **total\_chairs\_skiable\_ratio** | 0.00466667 |
| **fastQuads\_runs\_ratio** | 0.0285714 |
| **fastQuads\_skiable\_ratio** | 0.001 |



ski\_data.shape

(277, 36)



ski\_data = ski\_data[ski\_data.Name != 'Big Mountain Resort']



ski\_data.shape

(276, 36)

## 4.6 Train/Test Split

So far, you've treated ski resort data as a single entity. In machine learning, when you train your model on all of your data, you end up with no data set aside to evaluate model performance. You could keep making more and more complex models that fit the data better and better and not realise you were overfitting to that one set of samples. By partitioning the data into training and testing splits, without letting a model (or missing-value imputation) learn anything about the test split, you have a somewhat independent assessment of how your model might perform in the future. An often overlooked subtlety here is that people all too frequently use the test set to assess model performance and then compare multiple models to pick the best. This means their overall model selection process is fitting to one specific data set, now the test split. You could keep going, trying to get better and better performance on that one data set, but that's where cross-validation becomes especially useful. While training models, a test split is very useful as a final check on expected future performance.

What partition sizes would you have with a 70/30 train/test split?



len(ski\_data) \* .7, len(ski\_data) \* .3

(193.2, 82.8)



X\_train, X\_test, y\_train, y\_test = train\_test\_split(ski\_data.drop(columns='AdultWeekend'),

ski\_data.AdultWeekend, test\_size=0.3,

random\_state=47)



X\_train.shape, X\_test.shape

((193, 35), (83, 35))



y\_train.shape, y\_test.shape

((193,), (83,))



#Code task 1#

#Save the 'Name', 'state', and 'Region' columns from the train/test data into names\_train and names\_test

#Then drop those columns from `X\_train` and `X\_test`. Use 'inplace=True'

names\_list = ['Name', 'state', 'Region']

names\_train = X\_train[\_\_\_]

names\_test = X\_test[\_\_\_]

X\_train.\_\_\_(columns=names\_list, inplace=\_\_\_)

X\_test.\_\_\_(columns=names\_list, inplace=\_\_\_)

X\_train.shape, X\_test.shape



#Code task 2#

#Check the `dtypes` attribute of `X\_train` to verify all features are numeric

X\_train.\_\_\_



#Code task 3#

#Repeat this check for the test split in `X\_test`

X\_test.\_\_\_

You have only numeric features in your X now!

## 4.7 Initial Not-Even-A-Model

A good place to start is to see how good the mean is as a predictor. In other words, what if you simply say your best guess is the average price?



#Code task 4#

#Calculate the mean of `y\_train`

train\_mean = y\_train.\_\_\_

train\_mean

sklearn's DummyRegressor easily does this:



#Code task 5#

#Fit the dummy regressor on the training data

#Hint, call its `.fit()` method with `X\_train` and `y\_train` as arguments

#Then print the object's `constant\_` attribute and verify it's the same as the mean above

dumb\_reg = DummyRegressor(strategy='mean')

dumb\_reg.\_\_\_(\_\_\_, \_\_\_)

dumb\_reg.\_\_\_

How good is this? How closely does this match, or explain, the actual values? There are many ways of assessing how good one set of values agrees with another, which brings us to the subject of metrics.

### 4.7.1 Metrics

#### 4.7.1.1 R-squared, or coefficient of determination

One measure is *𝑅*2

, the [coefficient of determination](https://en.wikipedia.org/wiki/Coefficient_of_determination). This is a measure of the proportion of variance in the dependent variable (our ticket price) that is predicted by our "model". The linked Wikipedia articles gives a nice explanation of how negative values can arise. This is frequently a cause of confusion for newcomers who, reasonably, ask how can a squared value be negative?

Recall the mean can be denoted by *𝑦*¯

, where

*𝑦*¯=1*𝑛*∑*𝑖*=1*𝑛𝑦𝑖*

and where *𝑦𝑖*

are the individual values of the dependent variable.

The total sum of squares (error), can be expressed as

*𝑆𝑆𝑡𝑜𝑡*=∑*𝑖*(*𝑦𝑖*−*𝑦*¯)2

The above formula should be familiar as it's simply the variance without the denominator to scale (divide) by the sample size.

The residual sum of squares is similarly defined to be

*𝑆𝑆𝑟𝑒𝑠*=∑*𝑖*(*𝑦𝑖*−*𝑦*̂ )2

where *𝑦*̂

are our predicted values for the depended variable.

The coefficient of determination, *𝑅*2

, here is given by

*𝑅*2=1−*𝑆𝑆𝑟𝑒𝑠𝑆𝑆𝑡𝑜𝑡*

Putting it into words, it's one minus the ratio of the residual variance to the original variance. Thus, the baseline model here, which always predicts *𝑦*¯

, should give *𝑅*2=0. A model that perfectly predicts the observed values would have no residual error and so give *𝑅*2=1. Models that do worse than predicting the mean will have increased the sum of squares of residuals and so produce a negative *𝑅*2

.



#Code task 6#

#Calculate the R^2 as defined above

def r\_squared(y, ypred):

"""R-squared score.

Calculate the R-squared, or coefficient of determination, of the input.

Arguments:

y -- the observed values

ypred -- the predicted values

"""

ybar = np.sum(y) / len(y) #yes, we could use np.mean(y)

sum\_sq\_tot = np.\_\_\_((y - ybar)\*\*2) #total sum of squares error

sum\_sq\_res = np.\_\_\_((y - ypred)\*\*2) #residual sum of squares error

R2 = 1.0 - \_\_\_ / \_\_\_

return R2

Make your predictions by creating an array of length the size of the training set with the single value of the mean.



y\_tr\_pred\_ = train\_mean \* np.ones(len(y\_train))

y\_tr\_pred\_[:5]

array([63.81108808, 63.81108808, 63.81108808, 63.81108808, 63.81108808])

Remember the sklearn dummy regressor?



y\_tr\_pred = dumb\_reg.predict(X\_train)

y\_tr\_pred[:5]

array([63.81108808, 63.81108808, 63.81108808, 63.81108808, 63.81108808])

You can see that DummyRegressor produces exactly the same results and saves you having to mess about broadcasting the mean (or whichever other statistic we used - check out the [documentation](https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyRegressor.html) to see what's available) to an array of the appropriate length. It also gives you an object with fit() and predict() methods as well so you can use them as conveniently as any other sklearn estimator.



r\_squared(y\_train, y\_tr\_pred)

0.0

Exactly as expected, if you use the average value as your prediction, you get an *𝑅*2

of zero on our training set. What if you use this "model" to predict unseen values from the test set? Remember, of course, that your "model" is trained on the training set; you still use the training set mean as your prediction.

Make your predictions by creating an array of length the size of the test set with the single value of the (training) mean.



y\_te\_pred = train\_mean \* np.ones(len(y\_test))

r\_squared(y\_test, y\_te\_pred)

-0.0031235200417913944

Generally, you can expect performance on a test set to be slightly worse than on the training set. As you are getting an *𝑅*2

of zero on the training set, there's nowhere to go but negative!

*𝑅*2

is a common metric, and interpretable in terms of the amount of variance explained, it's less appealing if you want an idea of how "close" your predictions are to the true values. Metrics that summarise the difference between predicted and actual values are mean absolute error and mean squared error.

#### 4.7.1.2 Mean Absolute Error

This is very simply the average of the absolute errors:

*𝑀𝐴𝐸*=1*𝑛*∑*𝑖𝑛*|*𝑦𝑖*−*𝑦*̂ |



#Code task 7#

#Calculate the MAE as defined above

def mae(y, ypred):

"""Mean absolute error.

Calculate the mean absolute error of the arguments

​

Arguments:

y -- the observed values

ypred -- the predicted values

"""

abs\_error = np.abs(\_\_\_ - \_\_\_)

mae = np.mean(\_\_\_)

return mae



mae(y\_train, y\_tr\_pred)

17.923463717146785



mae(y\_test, y\_te\_pred)

19.136142081278486

Mean absolute error is arguably the most intuitive of all the metrics, this essentially tells you that, on average, you might expect to be off by around $19 if you guessed ticket price based on an average of known values.

#### 4.7.1.3 Mean Squared Error

Another common metric (and an important one internally for optimizing machine learning models) is the mean squared error. This is simply the average of the square of the errors:

*𝑀𝑆𝐸*=1*𝑛*∑*𝑖𝑛*(*𝑦𝑖*−*𝑦*̂ )2



#Code task 8#

#Calculate the MSE as defined above

def mse(y, ypred):

"""Mean square error.

Calculate the mean square error of the arguments

​

Arguments:

y -- the observed values

ypred -- the predicted values

"""

sq\_error = (\_\_\_ - \_\_\_)\*\*2

mse = np.mean(\_\_\_)

return mse



mse(y\_train, y\_tr\_pred)

614.1334096969057



mse(y\_test, y\_te\_pred)

581.4365441953481

So here, you get a slightly better MSE on the test set than you did on the train set. And what does a squared error mean anyway? To convert this back to our measurement space, we often take the square root, to form the root mean square error thus:



np.sqrt([mse(y\_train, y\_tr\_pred), mse(y\_test, y\_te\_pred)])

array([24.78171523, 24.11299534])

### 4.7.2 sklearn metrics

Functions are good, but you don't want to have to define functions every time we want to assess performance. sklearn.metrics provides many commonly used metrics, included the ones above.

##### 4.7.2.0.1 R-squared



r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_test, y\_te\_pred)

(0.0, -0.0031235200417913944)

##### 4.7.2.0.2 Mean absolute error



mean\_absolute\_error(y\_train, y\_tr\_pred), mean\_absolute\_error(y\_test, y\_te\_pred)

(17.92346371714677, 19.136142081278486)

##### 4.7.2.0.3 Mean squared error



mean\_squared\_error(y\_train, y\_tr\_pred), mean\_squared\_error(y\_test, y\_te\_pred)

(614.1334096969046, 581.4365441953483)

### 4.7.3 Note On Calculating Metrics

When calling functions to calculate metrics, it is important to take care in the order of the arguments. Two of the metrics above actually don't care if the arguments are reversed; one does. Which one cares?

In a Jupyter code cell, running r2\_score? will bring up the docstring for the function, and r2\_score?? will bring up the actual code of the function! Try them and compare the source for sklearn's function with yours. Feel free to explore what happens when you reverse the order of the arguments and compare behaviour of sklearn's function and yours.



# train set - sklearn

# correct order, incorrect order

r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_tr\_pred, y\_train)

(0.0, -3.041041349306602e+30)



# test set - sklearn

# correct order, incorrect order

r2\_score(y\_test, y\_te\_pred), r2\_score(y\_te\_pred, y\_test)

(-0.0031235200417913944, 0.0)



# train set - using our homebrew function

# correct order, incorrect order

r\_squared(y\_train, y\_tr\_pred), r\_squared(y\_tr\_pred, y\_train)

(0.0, -3.041041349306602e+30)



# test set - using our homebrew function

# correct order, incorrect order

r\_squared(y\_test, y\_te\_pred), r\_squared(y\_te\_pred, y\_test)

/home/guy/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:15: RuntimeWarning: divide by zero encountered in double\_scalars

from ipykernel import kernelapp as app

(-0.0031235200417913944, -inf)

You can get very different results swapping the argument order. It's worth highlighting this because data scientists do this too much in the real world! Don't be one of them! Frequently the argument order doesn't matter, but it will bite you when you do it with a function that does care. It's sloppy, bad practice and if you don't make a habit of putting arguments in the right order, you will forget!

Remember:

* argument order matters,
* check function syntax with func? in a code cell

## 4.8 Initial Models

### 4.8.1 Imputing missing feature (predictor) values

Recall when performing EDA, you imputed (filled in) some missing values in pandas. You did this judiciously for exploratory/visualization purposes. You left many missing values in the data. You can impute missing values using scikit-learn, but note that you should learn values to impute from a train split and apply that to the test split to then assess how well your imputation worked.

#### 4.8.1.1 Impute missing values with median

There's missing values. Recall from your data exploration that many distributions were skewed. Your first thought might be to impute missing values using the median.

##### 4.8.1.1.1 Learn the values to impute from the train set



# These are the values we'll use to fill in any missing values

X\_defaults\_median = X\_train.median()

X\_defaults\_median

summit\_elev 2215.000000

vertical\_drop 750.000000

base\_elev 1300.000000

trams 0.000000

fastSixes 0.000000

fastQuads 0.000000

quad 1.000000

triple 1.000000

double 1.000000

surface 2.000000

total\_chairs 7.000000

Runs 28.000000

TerrainParks 2.000000

LongestRun\_mi 1.000000

SkiableTerrain\_ac 170.000000

Snow Making\_ac 96.500000

daysOpenLastYear 109.000000

yearsOpen 57.000000

averageSnowfall 120.000000

projectedDaysOpen 115.000000

NightSkiing\_ac 70.000000

resorts\_per\_state 15.000000

resorts\_per\_100kcapita 0.248243

resorts\_per\_100ksq\_mile 22.902162

resort\_skiable\_area\_ac\_state\_ratio 0.051458

resort\_days\_open\_state\_ratio 0.071225

resort\_terrain\_park\_state\_ratio 0.069444

resort\_night\_skiing\_state\_ratio 0.077081

total\_chairs\_runs\_ratio 0.200000

total\_chairs\_skiable\_ratio 0.040323

fastQuads\_runs\_ratio 0.000000

fastQuads\_skiable\_ratio 0.000000

dtype: float64

##### 4.8.1.1.2 Apply the imputation to both train and test splits



#Code task 9#

#Call `X\_train` and `X\_test`'s `fillna()` method, passing `X\_defaults\_median` as the values to use

#Assign the results to `X\_tr` and `X\_te`, respectively

X\_tr = X\_train.\_\_\_(\_\_\_)

X\_te = X\_test.\_\_\_(\_\_\_)

##### 4.8.1.1.3 Scale the data

As you have features measured in many different units, with numbers that vary by orders of magnitude, start off by scaling them to put them all on a consistent scale. The [StandardScaler](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html) scales each feature to zero mean and unit variance.



#Code task 10#

#Call the StandardScaler`s fit method on `X\_tr` to fit the scaler

#then use it's `transform()` method to apply the scaling to both the train and test split

#data (`X\_tr` and `X\_te`), naming the results `X\_tr\_scaled` and `X\_te\_scaled`, respectively

scaler = StandardScaler()

scaler.\_\_\_(X\_tr)

X\_tr\_scaled = scaler.\_\_\_(X\_tr)

X\_te\_scaled = scaler.\_\_\_(X\_te)

##### 4.8.1.1.4 Train the model on the train split



lm = LinearRegression().fit(X\_tr\_scaled, y\_train)

##### 4.8.1.1.5 Make predictions using the model on both train and test splits



#Code task 11#

#Call the `predict()` method of the model (`lm`) on both the (scaled) train and test data

#Assign the predictions to `y\_tr\_pred` and `y\_te\_pred`, respectively

y\_tr\_pred = lm.\_\_\_(X\_tr\_scaled)

y\_te\_pred = lm.\_\_\_(X\_te\_scaled)

##### 4.8.1.1.6 Assess model performance



# r^2 - train, test

median\_r2 = r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_test, y\_te\_pred)

median\_r2

(0.8177988515690604, 0.7209725843435142)

Recall that you estimated ticket price by simply using a known average. As expected, this produced an *𝑅*2

of zero for both the training and test set, because *𝑅*2

tells us how much of the variance you're explaining beyond that of using just the mean, and you were using just the mean. Here we see that our simple linear regression model explains over 80% of the variance on the train set and over 70% on the test set. Clearly you are onto something, although the much lower value for the test set suggests you're overfitting somewhat. This isn't a surprise as you've made no effort to select a parsimonious set of features or deal with multicollinearity in our data.



#Code task 12#

#Now calculate the mean absolute error scores using `sklearn`'s `mean\_absolute\_error` function

# as we did above for R^2

# MAE - train, test

median\_mae = \_\_\_(y\_train, y\_tr\_pred), \_\_\_(y\_test, y\_te\_pred)

median\_mae

Using this model, then, on average you'd expect to estimate a ticket price within $9 or so of the real price. This is much, much better than the $19 from just guessing using the average. There may be something to this machine learning lark after all!



#Code task 13#

#And also do the same using `sklearn`'s `mean\_squared\_error`

# MSE - train, test

median\_mse = \_\_\_(\_\_\_, \_\_\_), \_\_\_(\_\_\_, \_\_\_)

median\_mse

#### 4.8.1.2 Impute missing values with the mean

You chose to use the median for filling missing values because of the skew of many of our predictor feature distributions. What if you wanted to try something else, such as the mean?

##### 4.8.1.2.1 Learn the values to impute from the train set



#Code task 14#

#As we did for the median above, calculate mean values for imputing missing values

# These are the values we'll use to fill in any missing values

X\_defaults\_mean = X\_train.\_\_\_()

X\_defaults\_mean

By eye, you can immediately tell that your replacement values are much higher than those from using the median.

##### 4.8.1.2.2 Apply the imputation to both train and test splits



X\_tr = X\_train.fillna(X\_defaults\_mean)

X\_te = X\_test.fillna(X\_defaults\_mean)

##### 4.8.1.2.3 Scale the data



scaler = StandardScaler()

scaler.fit(X\_tr)

X\_tr\_scaled = scaler.transform(X\_tr)

X\_te\_scaled = scaler.transform(X\_te)

##### 4.8.1.2.4 Train the model on the train split



lm = LinearRegression().fit(X\_tr\_scaled, y\_train)

##### 4.8.1.2.5 Make predictions using the model on both train and test splits



y\_tr\_pred = lm.predict(X\_tr\_scaled)

y\_te\_pred = lm.predict(X\_te\_scaled)

##### 4.8.1.2.6 Assess model performance



r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_test, y\_te\_pred)

(0.8170154093990025, 0.716381471695996)



mean\_absolute\_error(y\_train, y\_tr\_pred), mean\_absolute\_error(y\_test, y\_te\_pred)

(8.536884040670973, 9.416375625789271)



mean\_squared\_error(y\_train, y\_tr\_pred), mean\_squared\_error(y\_test, y\_te\_pred)

(112.37695054778276, 164.3926930952436)

These results don't seem very different to when you used the median for imputing missing values. Perhaps it doesn't make much difference here. Maybe your overtraining dominates. Maybe other feature transformations, such as taking the log, would help. You could try with just a subset of features rather than using all of them as inputs.

To perform the median/mean comparison, you copied and pasted a lot of code just to change the function for imputing missing values. It would make more sense to write a function that performed the sequence of steps:

1. impute missing values
2. scale the features
3. train a model
4. calculate model performance

But these are common steps and sklearn provides something much better than writing custom functions.

### 4.8.2 Pipelines

One of the most important and useful components of sklearn is the [pipeline](https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html). In place of panda's fillna DataFrame method, there is sklearn's SimpleImputer. Remember the first linear model above performed the steps:

1. replace missing values with the median for each feature
2. scale the data to zero mean and unit variance
3. train a linear regression model

and all these steps were trained on the train split and then applied to the test split for assessment.

The pipeline below defines exactly those same steps. Crucially, the resultant Pipeline object has a fit() method and a predict() method, just like the LinearRegression() object itself. Just as you might create a linear regression model and train it with .fit() and predict with .predict(), you can wrap the entire process of imputing and feature scaling and regression in a single object you can train with .fit() and predict with .predict(). And that's basically a pipeline: a model on steroids.

#### 4.8.2.1 Define the pipeline



pipe = make\_pipeline(

SimpleImputer(strategy='median'),

StandardScaler(),

LinearRegression()

)



type(pipe)

sklearn.pipeline.Pipeline



hasattr(pipe, 'fit'), hasattr(pipe, 'predict')

(True, True)

#### 4.8.2.2 Fit the pipeline

Here, a single call to the pipeline's fit() method combines the steps of learning the imputation (determining what values to use to fill the missing ones), the scaling (determining the mean to subtract and the variance to divide by), and then training the model. It does this all in the one call with the training data as arguments.



#Code task 15#

#Call the pipe's `fit()` method with `X\_train` and `y\_train` as arguments

pipe.\_\_\_(\_\_\_, \_\_\_)

#### 4.8.2.3 Make predictions on the train and test sets



y\_tr\_pred = pipe.predict(X\_train)

y\_te\_pred = pipe.predict(X\_test)

#### 4.8.2.4 Assess performance



r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_test, y\_te\_pred)

(0.8177988515690604, 0.7209725843435142)

And compare with your earlier (non-pipeline) result:



median\_r2

(0.8177988515690604, 0.7209725843435142)



mean\_absolute\_error(y\_train, y\_tr\_pred), mean\_absolute\_error(y\_test, y\_te\_pred)

(8.547850301825427, 9.40702011858132)



Compare with your earlier result:



median\_mae

(8.547850301825427, 9.40702011858132)



mean\_squared\_error(y\_train, y\_tr\_pred), mean\_squared\_error(y\_test, y\_te\_pred)

(111.89581253658478, 161.73156451192284)

Compare with your earlier result:



median\_mse

(111.89581253658478, 161.73156451192284)

These results confirm the pipeline is doing exactly what's expected, and results are identical to your earlier steps. This allows you to move faster but with confidence.

## 4.9 Refining The Linear Model

You suspected the model was overfitting. This is no real surprise given the number of features you blindly used. It's likely a judicious subset of features would generalize better. sklearn has a number of feature selection functions available. The one you'll use here is SelectKBest which, as you might guess, selects the k best features. You can read about SelectKBest [here](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html#sklearn.feature_selection.SelectKBest). f\_regression is just the [score function](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_regression.html#sklearn.feature_selection.f_regression) you're using because you're performing regression. It's important to choose an appropriate one for your machine learning task.

### 4.9.1 Define the pipeline

Redefine your pipeline to include this feature selection step:



#Code task 16#

#Add `SelectKBest` as a step in the pipeline between `StandardScaler()` and `LinearRegression()`

#Don't forget to tell it to use `f\_regression` as its score function

pipe = make\_pipeline(

SimpleImputer(strategy='median'),

StandardScaler(),

\_\_\_(\_\_\_),

LinearRegression()

)

### 4.9.2 Fit the pipeline



pipe.fit(X\_train, y\_train)

Pipeline(steps=[('simpleimputer', SimpleImputer(strategy='median')),

('standardscaler', StandardScaler()),

('selectkbest',

SelectKBest(score\_func=<function f\_regression at 0x7fc8cd9c8950>)),

('linearregression', LinearRegression())])

### 4.9.3 Assess performance on the train and test set



y\_tr\_pred = pipe.predict(X\_train)

y\_te\_pred = pipe.predict(X\_test)



r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_test, y\_te\_pred)

(0.7674914326052744, 0.6259877354190833)



mean\_absolute\_error(y\_train, y\_tr\_pred), mean\_absolute\_error(y\_test, y\_te\_pred)

(9.501495079727484, 11.201830190332057)

This has made things worse! Clearly selecting a subset of features has an impact on performance. SelectKBest defaults to k=10. You've just seen that 10 is worse than using all features. What is the best k? You could create a new pipeline with a different value of k:

### 4.9.4 Define a new pipeline to select a different number of features



#Code task 17#

#Modify the `SelectKBest` step to use a value of 15 for k

pipe15 = make\_pipeline(

SimpleImputer(strategy='median'),

StandardScaler(),

\_\_\_(\_\_\_, k=\_\_\_),

LinearRegression()

)

### 4.9.5 Fit the pipeline



pipe15.fit(X\_train, y\_train)

Pipeline(steps=[('simpleimputer', SimpleImputer(strategy='median')),

('standardscaler', StandardScaler()),

('selectkbest',

SelectKBest(k=15,

score\_func=<function f\_regression at 0x7fc8cd9c8950>)),

('linearregression', LinearRegression())])

### 4.9.6 Assess performance on train and test data



y\_tr\_pred = pipe15.predict(X\_train)

y\_te\_pred = pipe15.predict(X\_test)



r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_test, y\_te\_pred)

(0.7924096060483825, 0.6376199973170795)



mean\_absolute\_error(y\_train, y\_tr\_pred), mean\_absolute\_error(y\_test, y\_te\_pred)

(9.211767769307116, 10.488246867294356)

You could keep going, trying different values of k, training a model, measuring performance on the test set, and then picking the model with the best test set performance. There's a fundamental problem with this approach: you're tuning the model to the arbitrary test set! If you continue this way you'll end up with a model works well on the particular quirks of our test set but fails to generalize to new data. The whole point of keeping a test set is for it to be a set of that new data, to check how well our model might perform on data it hasn't seen.

The way around this is a technique called cross-validation. You partition the training set into k folds, train our model on k-1 of those folds, and calculate performance on the fold not used in training. This procedure then cycles through k times with a different fold held back each time. Thus you end up building k models on k sets of data with k estimates of how the model performs on unseen data but without having to touch the test set.

### 4.9.7 Assessing performance using cross-validation



cv\_results = cross\_validate(pipe15, X\_train, y\_train, cv=5)



cv\_scores = cv\_results['test\_score']

cv\_scores

array([0.63760862, 0.72831381, 0.74443537, 0.5487915 , 0.50441472])

Without using the same random state for initializing the CV folds, your actual numbers will be different.



np.mean(cv\_scores), np.std(cv\_scores)

(0.6327128053007867, 0.09502487849877672)

These results highlight that assessing model performance in inherently open to variability. You'll get different results depending on the quirks of which points are in which fold. An advantage of this is that you can also obtain an estimate of the variability, or uncertainty, in your performance estimate.



np.round((np.mean(cv\_scores) - 2 \* np.std(cv\_scores), np.mean(cv\_scores) + 2 \* np.std(cv\_scores)), 2)

array([0.44, 0.82])

### 4.9.8 Hyperparameter search using GridSearchCV

Pulling the above together, we have:

* a pipeline that
  + imputes missing values
  + scales the data
  + selects the k best features
  + trains a linear regression model
* a technique (cross-validation) for estimating model performance

Now you want to use cross-validation for multiple values of k and use cross-validation to pick the value of k that gives the best performance. make\_pipeline automatically names each step as the lowercase name of the step and the parameters of the step are then accessed by appending a double underscore followed by the parameter name. You know the name of the step will be 'selectkbest' and you know the parameter is 'k'.

You can also list the names of all the parameters in a pipeline like this:



#Code task 18#

#Call `pipe`'s `get\_params()` method to get a dict of available parameters and print their names

#using dict's `keys()` method

pipe.\_\_\_.keys()

The above can be particularly useful as your pipelines becomes more complex (you can even nest pipelines within pipelines).



k = [k+1 for k in range(len(X\_train.columns))]

grid\_params = {'selectkbest\_\_k': k}

Now you have a range of k to investigate. Is 1 feature best? 2? 3? 4? All of them? You could write a for loop and iterate over each possible value, doing all the housekeeping oyurselves to track the best value of k. But this is a common task so there's a built in function in sklearn. This is [GridSearchCV](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html). This takes the pipeline object, in fact it takes anything with a .fit() and .predict() method. In simple cases with no feature selection or imputation or feature scaling etc. you may see the classifier or regressor object itself directly passed into GridSearchCV. The other key input is the parameters and values to search over. Optional parameters include the cross-validation strategy and number of CPUs to use.



lr\_grid\_cv = GridSearchCV(pipe, param\_grid=grid\_params, cv=5, n\_jobs=-1)



lr\_grid\_cv.fit(X\_train, y\_train)

GridSearchCV(cv=5,

estimator=Pipeline(steps=[('simpleimputer',

SimpleImputer(strategy='median')),

('standardscaler', StandardScaler()),

('selectkbest',

SelectKBest(score\_func=<function f\_regression at 0x7fc8cd9c8950>)),

('linearregression',

LinearRegression())]),

n\_jobs=-1,

param\_grid={'selectkbest\_\_k': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,

12, 13, 14, 15, 16, 17, 18, 19, 20,

21, 22, 23, 24, 25, 26, 27, 28, 29,

30, ...]})



score\_mean = lr\_grid\_cv.cv\_results\_['mean\_test\_score']

score\_std = lr\_grid\_cv.cv\_results\_['std\_test\_score']

cv\_k = [k for k in lr\_grid\_cv.cv\_results\_['param\_selectkbest\_\_k']]



#Code task 19#

#Print the `best\_params\_` attribute of `lr\_grid\_cv`

lr\_grid\_cv.\_\_\_



#Code task 20#

#Assign the value of k from the above dict of `best\_params\_` and assign it to `best\_k`

\_\_\_ = lr\_grid\_cv.\_\_\_['selectkbest\_\_k']

plt.subplots(figsize=(10, 5))

plt.errorbar(cv\_k, score\_mean, yerr=score\_std)

plt.axvline(x=best\_k, c='r', ls='--', alpha=.5)

plt.xlabel('k')

plt.ylabel('CV score (r-squared)')

plt.title('Pipeline mean CV score (error bars +/- 1sd)');

The above suggests a good value for k is 8. There was an initial rapid increase with k, followed by a slow decline. Also noticeable is the variance of the results greatly increase above k=8. As you increasingly overfit, expect greater swings in performance as different points move in and out of the train/test folds.

Which features were most useful? Step into your best model, shown below. Starting with the fitted grid search object, you get the best estimator, then the named step 'selectkbest', for which you can its get\_support() method for a logical mask of the features selected.



selected = lr\_grid\_cv.best\_estimator\_.named\_steps.selectkbest.get\_support()

Similarly, instead of using the 'selectkbest' named step, you can access the named step for the linear regression model and, from that, grab the model coefficients via its coef\_ attribute:



#Code task 21#

#Get the linear model coefficients from the `coef\_` attribute and store in `coefs`,

#get the matching feature names from the column names of the dataframe,

#and display the results as a pandas Series with `coefs` as the values and `features` as the index,

#sorting the values in descending order

coefs = lr\_grid\_cv.best\_estimator\_.named\_steps.linearregression.coef\_

features = X\_train.columns[selected]

pd.Series(\_\_\_, index=\_\_\_).\_\_\_(ascending=\_\_\_)

These results suggest that vertical drop is your biggest positive feature. This makes intuitive sense and is consistent with what you saw during the EDA work. Also, you see the area covered by snow making equipment is a strong positive as well. People like guaranteed skiing! The skiable terrain area is negatively associated with ticket price! This seems odd. People will pay less for larger resorts? There could be all manner of reasons for this. It could be an effect whereby larger resorts can host more visitors at any one time and so can charge less per ticket. As has been mentioned previously, the data are missing information about visitor numbers. Bear in mind, the coefficient for skiable terrain is negative for this model. For example, if you kept the total number of chairs and fastQuads constant, but increased the skiable terrain extent, you might imagine the resort is worse off because the chairlift capacity is stretched thinner.

## 4.10 Random Forest Model

A model that can work very well in a lot of cases is the random forest. For regression, this is provided by sklearn's RandomForestRegressor class.

Time to stop the bad practice of repeatedly checking performance on the test split. Instead, go straight from defining the pipeline to assessing performance using cross-validation. cross\_validate will perform the fitting as part of the process. This uses the default settings for the random forest so you'll then proceed to investigate some different hyperparameters.

### 4.10.1 Define the pipeline



#Code task 22#

#Define a pipeline comprising the steps:

#SimpleImputer() with a strategy of 'median'

#StandardScaler(),

#and then RandomForestRegressor() with a random state of 47

RF\_pipe = make\_pipeline(

\_\_\_(strategy=\_\_\_),

\_\_\_,

\_\_\_(random\_state=\_\_\_)

)

### 4.10.2 Fit and assess performance using cross-validation



#Code task 23#

#Call `cross\_validate` to estimate the pipeline's performance.

#Pass it the random forest pipe object, `X\_train` and `y\_train`,

#and get it to use 5-fold cross-validation

rf\_default\_cv\_results = cross\_validate(\_\_\_, \_\_\_, \_\_\_, cv=\_\_\_)



rf\_cv\_scores = rf\_default\_cv\_results['test\_score']

rf\_cv\_scores

array([0.69249204, 0.78061953, 0.77546915, 0.62190924, 0.61742339])



np.mean(rf\_cv\_scores), np.std(rf\_cv\_scores)

(0.6975826707112506, 0.07090742940774528)

### 4.10.3 Hyperparameter search using GridSearchCV

Random forest has a number of hyperparameters that can be explored, however here you'll limit yourselves to exploring some different values for the number of trees. You'll try it with and without feature scaling, and try both the mean and median as strategies for imputing missing values.



n\_est = [int(n) for n in np.logspace(start=1, stop=3, num=20)]

grid\_params = {

'randomforestregressor\_\_n\_estimators': n\_est,

'standardscaler': [StandardScaler(), None],

'simpleimputer\_\_strategy': ['mean', 'median']

}

grid\_params

{'randomforestregressor\_\_n\_estimators': [10,

12,

16,

20,

26,

33,

42,

54,

69,

88,

112,

143,

183,

233,

297,

379,

483,

615,

784,

1000],

'standardscaler': [StandardScaler(), None],

'simpleimputer\_\_strategy': ['mean', 'median']}



#Code task 24#

#Call `GridSearchCV` with the random forest pipeline, passing in the above `grid\_params`

#dict for parameters to evaluate, 5-fold cross-validation, and all available CPU cores (if desired)

rf\_grid\_cv = GridSearchCV(\_\_\_, param\_grid=\_\_\_, cv=\_\_\_, n\_jobs=-1)



#Code task 25#

#Now call the `GridSearchCV`'s `fit()` method with `X\_train` and `y\_train` as arguments

#to actually start the grid search. This may take a minute or two.

rf\_grid\_cv.\_\_\_(\_\_\_, \_\_\_)



#Code task 26#

#Print the best params (`best\_params\_` attribute) from the grid search

rf\_grid\_cv.\_\_\_

It looks like imputing with the median helps, but scaling the features doesn't.



rf\_best\_cv\_results = cross\_validate(rf\_grid\_cv.best\_estimator\_, X\_train, y\_train, cv=5)

rf\_best\_scores = rf\_best\_cv\_results['test\_score']

rf\_best\_scores

array([0.6951357 , 0.79430697, 0.77170917, 0.62254707, 0.66499334])



np.mean(rf\_best\_scores), np.std(rf\_best\_scores)

(0.7097384501425082, 0.06451341966873386)

You've marginally improved upon the default CV results. Random forest has many more hyperparameters you could tune, but we won't dive into that here.



#Code task 27#

#Plot a barplot of the random forest's feature importances,

#assigning the `feature\_importances\_` attribute of

#`rf\_grid\_cv.best\_estimator\_.named\_steps.randomforestregressor` to the name `imps` to then

#create a pandas Series object of the feature importances, with the index given by the

#training data column names, sorting the values in descending order

plt.subplots(figsize=(10, 5))

imps = rf\_grid\_cv.best\_estimator\_.named\_steps.randomforestregressor.\_\_\_

rf\_feat\_imps = pd.Series(\_\_\_, index=X\_train.columns).sort\_values(ascending=False)

rf\_feat\_imps.plot(kind='bar')

plt.xlabel('features')

plt.ylabel('importance')

plt.title('Best random forest regressor feature importances');

Encouragingly, the dominant top four features are in common with your linear model:

* fastQuads
* Runs
* Snow Making\_ac
* vertical\_drop

## 4.11 Final Model Selection

Time to select your final model to use for further business modeling! It would be good to revisit the above model selection; there is undoubtedly more that could be done to explore possible hyperparameters. It would also be worthwhile to investigate removing the least useful features. Gathering or calculating, and storing, features adds business cost and dependencies, so if features genuinely are not needed they should be removed. Building a simpler model with fewer features can also have the advantage of being easier to sell (and/or explain) to stakeholders. Certainly there seem to be four strong features here and so a model using only those would probably work well. However, you want to explore some different scenarios where other features vary so keep the fuller model for now. The business is waiting for this model and you have something that you have confidence in to be much better than guessing with the average price.

Or, rather, you have two "somethings". You built a best linear model and a best random forest model. You need to finally choose between them. You can calculate the mean absolute error using cross-validation. Although cross-validate defaults to the *𝑅*2

[metric for scoring](https://scikit-learn.org/stable/modules/model_evaluation.html#scoring) regression, you can specify the mean absolute error as an alternative via the scoring parameter.

### 4.11.1 Linear regression model performance



# 'neg\_mean\_absolute\_error' uses the (negative of) the mean absolute error

lr\_neg\_mae = cross\_validate(lr\_grid\_cv.best\_estimator\_, X\_train, y\_train,

scoring='neg\_mean\_absolute\_error', cv=5, n\_jobs=-1)



lr\_mae\_mean = np.mean(-1 \* lr\_neg\_mae['test\_score'])

lr\_mae\_std = np.std(-1 \* lr\_neg\_mae['test\_score'])

lr\_mae\_mean, lr\_mae\_std

(10.499032338015297, 1.6220608976799646)



mean\_absolute\_error(y\_test, lr\_grid\_cv.best\_estimator\_.predict(X\_test))

11.793465668669327

### 4.11.2 Random forest regression model performance



rf\_neg\_mae = cross\_validate(rf\_grid\_cv.best\_estimator\_, X\_train, y\_train,

scoring='neg\_mean\_absolute\_error', cv=5, n\_jobs=-1)



rf\_mae\_mean = np.mean(-1 \* rf\_neg\_mae['test\_score'])

rf\_mae\_std = np.std(-1 \* rf\_neg\_mae['test\_score'])

rf\_mae\_mean, rf\_mae\_std

(9.644639167595688, 1.3528565172191818)



mean\_absolute\_error(y\_test, rf\_grid\_cv.best\_estimator\_.predict(X\_test))

9.537730050637332

### 4.11.3 Conclusion

The random forest model has a lower cross-validation mean absolute error by almost $1. It also exhibits less variability. Verifying performance on the test set produces performance consistent with the cross-validation results.

## 4.12 Data quantity assessment

Finally, you need to advise the business whether it needs to undertake further data collection. Would more data be useful? We're often led to believe more data is always good, but gathering data invariably has a cost associated with it. Assess this trade off by seeing how performance varies with differing data set sizes. The learning\_curve function does this conveniently.



fractions = [.2, .25, .3, .35, .4, .45, .5, .6, .75, .8, 1.0]

train\_size, train\_scores, test\_scores = learning\_curve(pipe, X\_train, y\_train, train\_sizes=fractions)

train\_scores\_mean = np.mean(train\_scores, axis=1)

train\_scores\_std = np.std(train\_scores, axis=1)

test\_scores\_mean = np.mean(test\_scores, axis=1)

test\_scores\_std = np.std(test\_scores, axis=1)



plt.subplots(figsize=(10, 5))

plt.errorbar(train\_size, test\_scores\_mean, yerr=test\_scores\_std)

plt.xlabel('Training set size')

plt.ylabel('CV scores')

plt.title('Cross-validation score as training set size increases');

A picture containing timeline

Description automatically generated

This shows that you seem to have plenty of data. There's an initial rapid improvement in model scores as one would expect, but it's essentially levelled off by around a sample size of 40-50.

## 4.13 Save best model object from pipeline



#Code task 28#

#This may not be "production grade ML deployment" practice, but adding some basic

#information to your saved models can save your bacon in development.

#Just what version model have you just loaded to reuse? What version of `sklearn`

#created it? When did you make it?

#Assign the pandas version number (`pd.\_\_version\_\_`) to the `pandas\_version` attribute,

#the numpy version (`np.\_\_version\_\_`) to the `numpy\_version` attribute,

#the sklearn version (`sklearn\_version`) to the `sklearn\_version` attribute,

#and the current datetime (`datetime.datetime.now()`) to the `build\_datetime` attribute

#Let's call this model version '1.0'

best\_model = rf\_grid\_cv.best\_estimator\_

best\_model.version = \_\_\_

best\_model.pandas\_version = \_\_\_

best\_model.numpy\_version = \_\_\_

best\_model.sklearn\_version = \_\_\_

best\_model.X\_columns = [col for col in X\_train.columns]

best\_model.build\_datetime = \_\_\_



# save the model

​

modelpath = '../models'

save\_file(best\_model, 'ski\_resort\_pricing\_model.pkl', modelpath)

## 4.14 Summary

**Q: 1** Write a summary of the work in this notebook. Capture the fact that you gained a baseline idea of performance by simply taking the average price and how well that did. Then highlight that you built a linear model and the features that found. Comment on the estimate of its performance from cross-validation and whether its performance on the test split was consistent with this estimate. Also highlight that a random forest regressor was tried, what preprocessing steps were found to be best, and again what its estimated performance via cross-validation was and whether its performance on the test set was consistent with that. State which model you have decided to use going forwards and why. This summary should provide a quick overview for someone wanting to know quickly why the given model was chosen for the next part of the business problem to help guide important business decisions.

**A: 1** Your answer here

[Jupyter Notebook](http://localhost:8890/tree?token=34101dfefd012e8bd0b705731d01b2a4049a6063cf628fdf)

04\_preprocessing\_and\_training Last Checkpoint: a minute ago (autosaved)

Python 3 (ipykernel)

* [File](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [Edit](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [View](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [Insert](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [Cell](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [Kernel](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [Widgets](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)
* [Help](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb)

# 4 Pre-Processing and Training Data

## 4.1 Contents

* [4 Pre-Processing and Training Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4_Pre-Processing_and_Training_Data)
  + [4.1 Contents](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.1_Contents)
  + [4.2 Introduction](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.2_Introduction)
  + [4.3 Imports](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.3_Imports)
  + [4.4 Load Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.4_Load_Data)
  + [4.5 Extract Big Mountain Data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.5_Extract_Big_Mountain_Data)
  + [4.6 Train/Test Split](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.6_Train/Test_Split)
  + [4.7 Initial Not-Even-A-Model](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7_Initial_Not-Even-A-Model)
    - [4.7.1 Metrics](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.1_Metrics)
      * [4.7.1.1 R-squared, or coefficient of determination](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.1.1_R-squared,_or_coefficient_of_determination)
      * [4.7.1.2 Mean Absolute Error](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.1.2_Mean_Absolute_Error)
      * [4.7.1.3 Mean Squared Error](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.1.3_Mean_Squared_Error)
    - [4.7.2 sklearn metrics](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.2_sklearn_metrics)
      * [4.7.2.0.1 R-squared](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.2.0.1_R-squared)
      * [4.7.2.0.2 Mean absolute error](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.2.0.2_Mean_absolute_error)
      * [4.7.2.0.3 Mean squared error](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.2.0.3_Mean_squared_error)
    - [4.7.3 Note On Calculating Metrics](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.7.3_Note_On_Calculating_Metrics)
  + [4.8 Initial Models](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8_Initial_Models)
    - [4.8.1 Imputing missing feature (predictor) values](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1_Imputing_missing_feature_(predictor)_values)
      * [4.8.1.1 Impute missing values with median](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1_Impute_missing_values_with_median)
        + [4.8.1.1.1 Learn the values to impute from the train set](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1.1_Learn_the_values_to_impute_from_the_train_set)
        + [4.8.1.1.2 Apply the imputation to both train and test splits](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1.2_Apply_the_imputation_to_both_train_and_test_splits)
        + [4.8.1.1.3 Scale the data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1.3_Scale_the_data)
        + [4.8.1.1.4 Train the model on the train split](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1.4_Train_the_model_on_the_train_split)
        + [4.8.1.1.5 Make predictions using the model on both train and test splits](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1.5_Make_predictions_using_the_model_on_both_train_and_test_splits)
        + [4.8.1.1.6 Assess model performance](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.1.6_Assess_model_performance)
      * [4.8.1.2 Impute missing values with the mean](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2_Impute_missing_values_with_the_mean)
        + [4.8.1.2.1 Learn the values to impute from the train set](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2.1_Learn_the_values_to_impute_from_the_train_set)
        + [4.8.1.2.2 Apply the imputation to both train and test splits](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2.2_Apply_the_imputation_to_both_train_and_test_splits)
        + [4.8.1.2.3 Scale the data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2.3_Scale_the_data)
        + [4.8.1.2.4 Train the model on the train split](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2.4_Train_the_model_on_the_train_split)
        + [4.8.1.2.5 Make predictions using the model on both train and test splits](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2.5_Make_predictions_using_the_model_on_both_train_and_test_splits)
        + [4.8.1.2.6 Assess model performance](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.1.2.6_Assess_model_performance)
    - [4.8.2 Pipelines](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.2_Pipelines)
      * [4.8.2.1 Define the pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.2.1_Define_the_pipeline)
      * [4.8.2.2 Fit the pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.2.2_Fit_the_pipeline)
      * [4.8.2.3 Make predictions on the train and test sets](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.2.3_Make_predictions_on_the_train_and_test_sets)
      * [4.8.2.4 Assess performance](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.8.2.4_Assess_performance)
  + [4.9 Refining The Linear Model](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9_Refining_The_Linear_Model)
    - [4.9.1 Define the pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.1_Define_the_pipeline)
    - [4.9.2 Fit the pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.2_Fit_the_pipeline)
    - [4.9.3 Assess performance on the train and test set](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.3_Assess_performance_on_the_train_and_test_set)
    - [4.9.4 Define a new pipeline to select a different number of features](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.4_Define_a_new_pipeline_to_select_a_different_number_of_features)
    - [4.9.5 Fit the pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.5_Fit_the_pipeline)
    - [4.9.6 Assess performance on train and test data](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.6_Assess_performance_on_train_and_test_data)
    - [4.9.7 Assessing performance using cross-validation](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.7_Assessing_performance_using_cross-validation)
    - [4.9.8 Hyperparameter search using GridSearchCV](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.9.8_Hyperparameter_search_using_GridSearchCV)
  + [4.10 Random Forest Model](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.10_Random_Forest_Model)
    - [4.10.1 Define the pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.10.1_Define_the_pipeline)
    - [4.10.2 Fit and assess performance using cross-validation](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.10.2_Fit_and_assess_performance_using_cross-validation)
    - [4.10.3 Hyperparameter search using GridSearchCV](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.10.3_Hyperparameter_search_using_GridSearchCV)
  + [4.11 Final Model Selection](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.11_Final_Model_Selection)
    - [4.11.1 Linear regression model performance](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.11.1_Linear_regression_model_performance)
    - [4.11.2 Random forest regression model performance](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.11.2_Random_forest_regression_model_performance)
    - [4.11.3 Conclusion](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.11.3_Conclusion)
  + [4.12 Data quantity assessment](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.12_Data_quantity_assessment)
  + [4.13 Save best model object from pipeline](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.13_Save_best_model_object_from_pipeline)
  + [4.14 Summary](http://localhost:8890/notebooks/Desktop/Springboard/GitHub/Springboard2/Notebooks/04_preprocessing_and_training.ipynb#4.14_Summary)

## 4.2 Introduction

In preceding notebooks, performed preliminary assessments of data quality and refined the question to be answered. You found a small number of data values that gave clear choices about whether to replace values or drop a whole row. You determined that predicting the adult weekend ticket price was your primary aim. You threw away records with missing price data, but not before making the most of the other available data to look for any patterns between the states. You didn't see any and decided to treat all states equally; the state label didn't seem to be particularly useful.

In this notebook you'll start to build machine learning models. Before even starting with learning a machine learning model, however, start by considering how useful the mean value is as a predictor. This is more than just a pedagogical device. You never want to go to stakeholders with a machine learning model only to have the CEO point out that it performs worse than just guessing the average! Your first model is a baseline performance comparitor for any subsequent model. You then build up the process of efficiently and robustly creating and assessing models against it. The development we lay out may be little slower than in the real world, but this step of the capstone is definitely more than just instructional. It is good practice to build up an understanding that the machine learning pipelines you build work as expected. You can validate steps with your own functions for checking expected equivalence between, say, pandas and sklearn implementations.

## 4.3 Imports



import pandas as pd

import numpy as np

import os

import pickle

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn import \_\_version\_\_ as sklearn\_version

from sklearn.decomposition import PCA

from sklearn.preprocessing import scale

from sklearn.model\_selection import train\_test\_split, cross\_validate, GridSearchCV, learning\_curve

from sklearn.preprocessing import StandardScaler, MinMaxScaler

from sklearn.dummy import DummyRegressor

from sklearn.linear\_model import LinearRegression

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import r2\_score, mean\_squared\_error, mean\_absolute\_error

from sklearn.pipeline import make\_pipeline

from sklearn.impute import SimpleImputer

from sklearn.feature\_selection import SelectKBest, f\_regression

import datetime

​

from library.sb\_utils import save\_file

## 4.4 Load Data



ski\_data = pd.read\_csv('../data/ski\_data\_step3\_features.csv')

ski\_data.head().T

|  | **0** | **1** | **2** | **3** | **4** |
| --- | --- | --- | --- | --- | --- |
| **Name** | Alyeska Resort | Eaglecrest Ski Area | Hilltop Ski Area | Arizona Snowbowl | Sunrise Park Resort |
| **Region** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **state** | Alaska | Alaska | Alaska | Arizona | Arizona |
| **summit\_elev** | 3939 | 2600 | 2090 | 11500 | 11100 |
| **vertical\_drop** | 2500 | 1540 | 294 | 2300 | 1800 |
| **base\_elev** | 250 | 1200 | 1796 | 9200 | 9200 |
| **trams** | 1 | 0 | 0 | 0 | 0 |
| **fastSixes** | 0 | 0 | 0 | 1 | 0 |
| **fastQuads** | 2 | 0 | 0 | 0 | 1 |
| **quad** | 2 | 0 | 0 | 2 | 2 |
| **triple** | 0 | 0 | 1 | 2 | 3 |
| **double** | 0 | 4 | 0 | 1 | 1 |
| **surface** | 2 | 0 | 2 | 2 | 0 |
| **total\_chairs** | 7 | 4 | 3 | 8 | 7 |
| **Runs** | 76.0 | 36.0 | 13.0 | 55.0 | 65.0 |
| **TerrainParks** | 2.0 | 1.0 | 1.0 | 4.0 | 2.0 |
| **LongestRun\_mi** | 1.0 | 2.0 | 1.0 | 2.0 | 1.2 |
| **SkiableTerrain\_ac** | 1610.0 | 640.0 | 30.0 | 777.0 | 800.0 |
| **Snow Making\_ac** | 113.0 | 60.0 | 30.0 | 104.0 | 80.0 |
| **daysOpenLastYear** | 150.0 | 45.0 | 150.0 | 122.0 | 115.0 |
| **yearsOpen** | 60.0 | 44.0 | 36.0 | 81.0 | 49.0 |
| **averageSnowfall** | 669.0 | 350.0 | 69.0 | 260.0 | 250.0 |
| **AdultWeekend** | 85.0 | 53.0 | 34.0 | 89.0 | 78.0 |
| **projectedDaysOpen** | 150.0 | 90.0 | 152.0 | 122.0 | 104.0 |
| **NightSkiing\_ac** | 550.0 | NaN | 30.0 | NaN | 80.0 |
| **resorts\_per\_state** | 3 | 3 | 3 | 2 | 2 |
| **resorts\_per\_100kcapita** | 0.410091 | 0.410091 | 0.410091 | 0.027477 | 0.027477 |
| **resorts\_per\_100ksq\_mile** | 0.450867 | 0.450867 | 0.450867 | 1.75454 | 1.75454 |
| **resort\_skiable\_area\_ac\_state\_ratio** | 0.70614 | 0.280702 | 0.013158 | 0.492708 | 0.507292 |
| **resort\_days\_open\_state\_ratio** | 0.434783 | 0.130435 | 0.434783 | 0.514768 | 0.485232 |
| **resort\_terrain\_park\_state\_ratio** | 0.5 | 0.25 | 0.25 | 0.666667 | 0.333333 |
| **resort\_night\_skiing\_state\_ratio** | 0.948276 | NaN | 0.051724 | NaN | 1.0 |
| **total\_chairs\_runs\_ratio** | 0.092105 | 0.111111 | 0.230769 | 0.145455 | 0.107692 |
| **total\_chairs\_skiable\_ratio** | 0.004348 | 0.00625 | 0.1 | 0.010296 | 0.00875 |
| **fastQuads\_runs\_ratio** | 0.026316 | 0.0 | 0.0 | 0.0 | 0.015385 |
| **fastQuads\_skiable\_ratio** | 0.001242 | 0.0 | 0.0 | 0.0 | 0.00125 |

## 4.5 Extract Big Mountain Data

Big Mountain is your resort. Separate it from the rest of the data to use later.



big\_mountain = ski\_data[ski\_data.Name == 'Big Mountain Resort']



big\_mountain.T

|  | **124** |
| --- | --- |
| **Name** | Big Mountain Resort |
| **Region** | Montana |
| **state** | Montana |
| **summit\_elev** | 6817 |
| **vertical\_drop** | 2353 |
| **base\_elev** | 4464 |
| **trams** | 0 |
| **fastSixes** | 0 |
| **fastQuads** | 3 |
| **quad** | 2 |
| **triple** | 6 |
| **double** | 0 |
| **surface** | 3 |
| **total\_chairs** | 14 |
| **Runs** | 105.0 |
| **TerrainParks** | 4.0 |
| **LongestRun\_mi** | 3.3 |
| **SkiableTerrain\_ac** | 3000.0 |
| **Snow Making\_ac** | 600.0 |
| **daysOpenLastYear** | 123.0 |
| **yearsOpen** | 72.0 |
| **averageSnowfall** | 333.0 |
| **AdultWeekend** | 81.0 |
| **projectedDaysOpen** | 123.0 |
| **NightSkiing\_ac** | 600.0 |
| **resorts\_per\_state** | 12 |
| **resorts\_per\_100kcapita** | 1.122778 |
| **resorts\_per\_100ksq\_mile** | 8.161045 |
| **resort\_skiable\_area\_ac\_state\_ratio** | 0.140121 |
| **resort\_days\_open\_state\_ratio** | 0.129338 |
| **resort\_terrain\_park\_state\_ratio** | 0.148148 |
| **resort\_night\_skiing\_state\_ratio** | 0.84507 |
| **total\_chairs\_runs\_ratio** | 0.133333 |
| **total\_chairs\_skiable\_ratio** | 0.004667 |
| **fastQuads\_runs\_ratio** | 0.028571 |
| **fastQuads\_skiable\_ratio** | 0.001 |



ski\_data.shape

(277, 36)



ski\_data = ski\_data[ski\_data.Name != 'Big Mountain Resort']



ski\_data.shape

(276, 36)

## 4.6 Train/Test Split

So far, you've treated ski resort data as a single entity. In machine learning, when you train your model on all of your data, you end up with no data set aside to evaluate model performance. You could keep making more and more complex models that fit the data better and better and not realise you were overfitting to that one set of samples. By partitioning the data into training and testing splits, without letting a model (or missing-value imputation) learn anything about the test split, you have a somewhat independent assessment of how your model might perform in the future. An often overlooked subtlety here is that people all too frequently use the test set to assess model performance and then compare multiple models to pick the best. This means their overall model selection process is fitting to one specific data set, now the test split. You could keep going, trying to get better and better performance on that one data set, but that's where cross-validation becomes especially useful. While training models, a test split is very useful as a final check on expected future performance.

What partition sizes would you have with a 70/30 train/test split?



len(ski\_data) \* .7, len(ski\_data) \* .3

(193.2, 82.8)



X\_train, X\_test, y\_train, y\_test = train\_test\_split(ski\_data.drop(columns='AdultWeekend'),

ski\_data.AdultWeekend, test\_size=0.3,

random\_state=47)



X\_train.shape, X\_test.shape

((193, 35), (83, 35))



y\_train.shape, y\_test.shape

((193,), (83,))



#Code task 1#

#Save the 'Name', 'state', and 'Region' columns from the train/test data into names\_train and names\_test

#Then drop those columns from `X\_train` and `X\_test`. Use 'inplace=True'

names\_list = ['Name', 'state', 'Region']

names\_train = X\_train[names\_list]

names\_test = X\_test[names\_list]

X\_train.drop(columns=names\_list, inplace=True)

X\_test.drop(columns=names\_list, inplace=True)

X\_train.shape, X\_test.shape

((193, 32), (83, 32))



#Code task 2#

#Check the `dtypes` attribute of `X\_train` to verify all features are numeric

X\_train.dtypes

summit\_elev int64

vertical\_drop int64

base\_elev int64

trams int64

fastSixes int64

fastQuads int64

quad int64

triple int64

double int64

surface int64

total\_chairs int64

Runs float64

TerrainParks float64

LongestRun\_mi float64

SkiableTerrain\_ac float64

Snow Making\_ac float64

daysOpenLastYear float64

yearsOpen float64

averageSnowfall float64

projectedDaysOpen float64

NightSkiing\_ac float64

resorts\_per\_state int64

resorts\_per\_100kcapita float64

resorts\_per\_100ksq\_mile float64

resort\_skiable\_area\_ac\_state\_ratio float64

resort\_days\_open\_state\_ratio float64

resort\_terrain\_park\_state\_ratio float64

resort\_night\_skiing\_state\_ratio float64

total\_chairs\_runs\_ratio float64

total\_chairs\_skiable\_ratio float64

fastQuads\_runs\_ratio float64

fastQuads\_skiable\_ratio float64

dtype: object



#Code task 3#

#Repeat this check for the test split in `X\_test`

X\_test.dtypes

summit\_elev int64

vertical\_drop int64

base\_elev int64

trams int64

fastSixes int64

fastQuads int64

quad int64

triple int64

double int64

surface int64

total\_chairs int64

Runs float64

TerrainParks float64

LongestRun\_mi float64

SkiableTerrain\_ac float64

Snow Making\_ac float64

daysOpenLastYear float64

yearsOpen float64

averageSnowfall float64

projectedDaysOpen float64

NightSkiing\_ac float64

resorts\_per\_state int64

resorts\_per\_100kcapita float64

resorts\_per\_100ksq\_mile float64

resort\_skiable\_area\_ac\_state\_ratio float64

resort\_days\_open\_state\_ratio float64

resort\_terrain\_park\_state\_ratio float64

resort\_night\_skiing\_state\_ratio float64

total\_chairs\_runs\_ratio float64

total\_chairs\_skiable\_ratio float64

fastQuads\_runs\_ratio float64

fastQuads\_skiable\_ratio float64

dtype: object

You have only numeric features in your X now!

## 4.7 Initial Not-Even-A-Model

A good place to start is to see how good the mean is as a predictor. In other words, what if you simply say your best guess is the average price?



#Code task 4#

#Calculate the mean of `y\_train`

train\_mean = y\_train.mean()

train\_mean

63.811088082901556

sklearn's DummyRegressor easily does this:



#Code task 5#

#Fit the dummy regressor on the training data

#Hint, call its `.fit()` method with `X\_train` and `y\_train` as arguments

#Then print the object's `constant\_` attribute and verify it's the same as the mean above

dumb\_reg = DummyRegressor(strategy='mean')

dumb\_reg.fit(X\_train, y\_train)

dumb\_reg.constant\_

array([[63.81108808]])

How good is this? How closely does this match, or explain, the actual values? There are many ways of assessing how good one set of values agrees with another, which brings us to the subject of metrics.

### 4.7.1 Metrics

#### 4.7.1.1 R-squared, or coefficient of determination

One measure is *𝑅*2

, the [coefficient of determination](https://en.wikipedia.org/wiki/Coefficient_of_determination). This is a measure of the proportion of variance in the dependent variable (our ticket price) that is predicted by our "model". The linked Wikipedia articles gives a nice explanation of how negative values can arise. This is frequently a cause of confusion for newcomers who, reasonably, ask how can a squared value be negative?

Recall the mean can be denoted by *𝑦*¯

, where

*𝑦*¯=1*𝑛*∑*𝑖*=1*𝑛𝑦𝑖*

and where *𝑦𝑖*

are the individual values of the dependent variable.

The total sum of squares (error), can be expressed as

*𝑆𝑆𝑡𝑜𝑡*=∑*𝑖*(*𝑦𝑖*−*𝑦*¯)2

The above formula should be familiar as it's simply the variance without the denominator to scale (divide) by the sample size.

The residual sum of squares is similarly defined to be

*𝑆𝑆𝑟𝑒𝑠*=∑*𝑖*(*𝑦𝑖*−*𝑦*̂ )2

where *𝑦*̂

are our predicted values for the depended variable.

The coefficient of determination, *𝑅*2

, here is given by

*𝑅*2=1−*𝑆𝑆𝑟𝑒𝑠𝑆𝑆𝑡𝑜𝑡*

Putting it into words, it's one minus the ratio of the residual variance to the original variance. Thus, the baseline model here, which always predicts *𝑦*¯

, should give *𝑅*2=0. A model that perfectly predicts the observed values would have no residual error and so give *𝑅*2=1. Models that do worse than predicting the mean will have increased the sum of squares of residuals and so produce a negative *𝑅*2

.



#Code task 6#

#Calculate the R^2 as defined above

def r\_squared(y, ypred):

"""R-squared score.

Calculate the R-squared, or coefficient of determination, of the input.

Arguments:

y -- the observed values

ypred -- the predicted values

"""

ybar = np.sum(y) / len(y) #yes, we could use np.mean(y)

sum\_sq\_tot = np.sum((y - ybar)\*\*2) #total sum of squares error

sum\_sq\_res = np.sum((y - ypred)\*\*2) #residual sum of squares error

R2 = 1.0 - sum\_sq\_res / sum\_sq\_tot

return R2

Make your predictions by creating an array of length the size of the training set with the single value of the mean.



y\_tr\_pred\_ = train\_mean \* np.ones(len(y\_train))

y\_tr\_pred\_[:5]

array([63.81108808, 63.81108808, 63.81108808, 63.81108808, 63.81108808])

Remember the sklearn dummy regressor?



y\_tr\_pred = dumb\_reg.predict(X\_train)

y\_tr\_pred[:5]

array([63.81108808, 63.81108808, 63.81108808, 63.81108808, 63.81108808])

You can see that DummyRegressor produces exactly the same results and saves you having to mess about broadcasting the mean (or whichever other statistic we used - check out the [documentation](https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyRegressor.html) to see what's available) to an array of the appropriate length. It also gives you an object with fit() and predict() methods as well so you can use them as conveniently as any other sklearn estimator.



r\_squared(y\_train, y\_tr\_pred)

0.0

Exactly as expected, if you use the average value as your prediction, you get an *𝑅*2

of zero on our training set. What if you use this "model" to predict unseen values from the test set? Remember, of course, that your "model" is trained on the training set; you still use the training set mean as your prediction.

Make your predictions by creating an array of length the size of the test set with the single value of the (training) mean.



y\_te\_pred = train\_mean \* np.ones(len(y\_test))

r\_squared(y\_test, y\_te\_pred)

-0.0031235200417913944

Generally, you can expect performance on a test set to be slightly worse than on the training set. As you are getting an *𝑅*2

of zero on the training set, there's nowhere to go but negative!

*𝑅*2

is a common metric, and interpretable in terms of the amount of variance explained, it's less appealing if you want an idea of how "close" your predictions are to the true values. Metrics that summarise the difference between predicted and actual values are mean absolute error and mean squared error.

#### 4.7.1.2 Mean Absolute Error

This is very simply the average of the absolute errors:

*𝑀𝐴𝐸*=1*𝑛*∑*𝑖𝑛*|*𝑦𝑖*−*𝑦*̂ |



#Code task 7#

#Calculate the MAE as defined above

def mae(y, ypred):

"""Mean absolute error.

Calculate the mean absolute error of the arguments

​

Arguments:

y -- the observed values

ypred -- the predicted values

"""

abs\_error = np.abs(y - ypred)

mae = np.mean(abs\_error)

return mae



mae(y\_train, y\_tr\_pred)

17.923463717146785



mae(y\_test, y\_te\_pred)

19.136142081278486

Mean absolute error is arguably the most intuitive of all the metrics, this essentially tells you that, on average, you might expect to be off by around $19 if you guessed ticket price based on an average of known values.

#### 4.7.1.3 Mean Squared Error

Another common metric (and an important one internally for optimizing machine learning models) is the mean squared error. This is simply the average of the square of the errors:

*𝑀𝑆𝐸*=1*𝑛*∑*𝑖𝑛*(*𝑦𝑖*−*𝑦*̂ )2



#Code task 8#

#Calculate the MSE as defined above

def mse(y, ypred):

"""Mean square error.

Calculate the mean square error of the arguments

​

Arguments:

y -- the observed values

ypred -- the predicted values

"""

sq\_error = (y - ypred)\*\*2

mse = np.mean(sq\_error)

return mse



mse(y\_train, y\_tr\_pred)

614.1334096969057



mse(y\_test, y\_te\_pred)

581.4365441953481

So here, you get a slightly better MSE on the test set than you did on the train set. And what does a squared error mean anyway? To convert this back to our measurement space, we often take the square root, to form the root mean square error thus:



np.sqrt([mse(y\_train, y\_tr\_pred), mse(y\_test, y\_te\_pred)])

array([24.78171523, 24.11299534])

### 4.7.2 sklearn metrics

Functions are good, but you don't want to have to define functions every time we want to assess performance. sklearn.metrics provides many commonly used metrics, included the ones above.

##### 4.7.2.0.1 R-squared



r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_test, y\_te\_pred)

(0.0, -0.0031235200417913944)

##### 4.7.2.0.2 Mean absolute error



mean\_absolute\_error(y\_train, y\_tr\_pred), mean\_absolute\_error(y\_test, y\_te\_pred)

(17.92346371714677, 19.136142081278486)

##### 4.7.2.0.3 Mean squared error



mean\_squared\_error(y\_train, y\_tr\_pred), mean\_squared\_error(y\_test, y\_te\_pred)

(614.1334096969046, 581.4365441953483)

### 4.7.3 Note On Calculating Metrics

When calling functions to calculate metrics, it is important to take care in the order of the arguments. Two of the metrics above actually don't care if the arguments are reversed; one does. Which one cares?

In a Jupyter code cell, running r2\_score? will bring up the docstring for the function, and r2\_score?? will bring up the actual code of the function! Try them and compare the source for sklearn's function with yours. Feel free to explore what happens when you reverse the order of the arguments and compare behaviour of sklearn's function and yours.



# train set - sklearn

# correct order, incorrect order

r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_tr\_pred, y\_train)

(0.0, -3.041041349306602e+30)



# test set - sklearn

# correct order, incorrect order

r2\_score(y\_test, y\_te\_pred), r2\_score(y\_te\_pred, y\_test)

(-0.0031235200417913944, 0.0)



# train set - using our homebrew function

# correct order, incorrect order

r\_squared(y\_train, y\_tr\_pred), r\_squared(y\_tr\_pred, y\_train)

(0.0, -3.041041349306602e+30)



# test set - using our homebrew function

# correct order, incorrect order

r\_squared(y\_test, y\_te\_pred), r\_squared(y\_te\_pred, y\_test)

C:\Users\charrie\AppData\Local\Temp\ipykernel\_5300\1803819837.py:15: RuntimeWarning: divide by zero encountered in double\_scalars

R2 = 1.0 - sum\_sq\_res / sum\_sq\_tot

(-0.0031235200417913944, -inf)

You can get very different results swapping the argument order. It's worth highlighting this because data scientists do this too much in the real world! Don't be one of them! Frequently the argument order doesn't matter, but it will bite you when you do it with a function that does care. It's sloppy, bad practice and if you don't make a habit of putting arguments in the right order, you will forget!

Remember:

* argument order matters,
* check function syntax with func? in a code cell

## 4.8 Initial Models

### 4.8.1 Imputing missing feature (predictor) values

Recall when performing EDA, you imputed (filled in) some missing values in pandas. You did this judiciously for exploratory/visualization purposes. You left many missing values in the data. You can impute missing values using scikit-learn, but note that you should learn values to impute from a train split and apply that to the test split to then assess how well your imputation worked.

#### 4.8.1.1 Impute missing values with median

There's missing values. Recall from your data exploration that many distributions were skewed. Your first thought might be to impute missing values using the median.

##### 4.8.1.1.1 Learn the values to impute from the train set



# These are the values we'll use to fill in any missing values

X\_defaults\_median = X\_train.median()

X\_defaults\_median

summit\_elev 2215.000000

vertical\_drop 750.000000

base\_elev 1300.000000

trams 0.000000

fastSixes 0.000000

fastQuads 0.000000

quad 1.000000

triple 1.000000

double 1.000000

surface 2.000000

total\_chairs 7.000000

Runs 28.000000

TerrainParks 2.000000

LongestRun\_mi 1.000000

SkiableTerrain\_ac 170.000000

Snow Making\_ac 96.500000

daysOpenLastYear 109.000000

yearsOpen 57.000000

averageSnowfall 120.000000

projectedDaysOpen 115.000000

NightSkiing\_ac 70.000000

resorts\_per\_state 15.000000

resorts\_per\_100kcapita 0.248243

resorts\_per\_100ksq\_mile 22.902162

resort\_skiable\_area\_ac\_state\_ratio 0.051458

resort\_days\_open\_state\_ratio 0.071225

resort\_terrain\_park\_state\_ratio 0.069444

resort\_night\_skiing\_state\_ratio 0.077081

total\_chairs\_runs\_ratio 0.200000

total\_chairs\_skiable\_ratio 0.040323

fastQuads\_runs\_ratio 0.000000

fastQuads\_skiable\_ratio 0.000000

dtype: float64

##### 4.8.1.1.2 Apply the imputation to both train and test splits



#Code task 9#

#Call `X\_train` and `X\_test`'s `fillna()` method, passing `X\_defaults\_median` as the values to use

#Assign the results to `X\_tr` and `X\_te`, respectively

X\_tr = X\_train.fillna(X\_defaults\_median)

X\_te = X\_test.fillna(X\_defaults\_median)

##### 4.8.1.1.3 Scale the data

As you have features measured in many different units, with numbers that vary by orders of magnitude, start off by scaling them to put them all on a consistent scale. The [StandardScaler](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html) scales each feature to zero mean and unit variance.



#Code task 10#

#Call the StandardScaler`s fit method on `X\_tr` to fit the scaler

#then use it's `transform()` method to apply the scaling to both the train and test split

#data (`X\_tr` and `X\_te`), naming the results `X\_tr\_scaled` and `X\_te\_scaled`, respectively

scaler = StandardScaler()

scaler.fit(X\_tr)

X\_tr\_scaled = scaler.transform(X\_tr)

X\_te\_scaled = scaler.transform(X\_te)

##### 4.8.1.1.4 Train the model on the train split



lm = LinearRegression().fit(X\_tr\_scaled, y\_train)

##### 4.8.1.1.5 Make predictions using the model on both train and test splits



#Code task 11#

#Call the `predict()` method of the model (`lm`) on both the (scaled) train and test data

#Assign the predictions to `y\_tr\_pred` and `y\_te\_pred`, respectively

y\_tr\_pred = lm.predict(X\_tr\_scaled)

y\_te\_pred = lm.predict(X\_te\_scaled)

##### 4.8.1.1.6 Assess model performance



# r^2 - train, test

median\_r2 = r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_test, y\_te\_pred)

median\_r2

(0.8177988515690604, 0.7209725843435146)

Recall that you estimated ticket price by simply using a known average. As expected, this produced an *𝑅*2

of zero for both the training and test set, because *𝑅*2

tells us how much of the variance you're explaining beyond that of using just the mean, and you were using just the mean. Here we see that our simple linear regression model explains over 80% of the variance on the train set and over 70% on the test set. Clearly you are onto something, although the much lower value for the test set suggests you're overfitting somewhat. This isn't a surprise as you've made no effort to select a parsimonious set of features or deal with multicollinearity in our data.



#Code task 12#

#Now calculate the mean absolute error scores using `sklearn`'s `mean\_absolute\_error` function

# as we did above for R^2

# MAE - train, test

median\_mae = mean\_absolute\_error(y\_train, y\_tr\_pred), mean\_absolute\_error(y\_test, y\_te\_pred)

median\_mae

(8.547850301825429, 9.407020118581318)

Using this model, then, on average you'd expect to estimate a ticket price within $9 or so of the real price. This is much, much better than the $19 from just guessing using the average. There may be something to this machine learning lark after all!



#Code task 13#

#And also do the same using `sklearn`'s `mean\_squared\_error`

# MSE - train, test

median\_mse = mean\_squared\_error(y\_train, y\_tr\_pred), mean\_squared\_error(y\_test, y\_te\_pred)

median\_mse

(111.89581253658478, 161.73156451192264)

#### 4.8.1.2 Impute missing values with the mean

You chose to use the median for filling missing values because of the skew of many of our predictor feature distributions. What if you wanted to try something else, such as the mean?

##### 4.8.1.2.1 Learn the values to impute from the train set



#Code task 14#

#As we did for the median above, calculate mean values for imputing missing values

# These are the values we'll use to fill in any missing values

X\_defaults\_mean = X\_train.mean()

X\_defaults\_mean

summit\_elev 4074.554404

vertical\_drop 1043.196891

base\_elev 3020.512953

trams 0.103627

fastSixes 0.072539

fastQuads 0.673575

quad 1.010363

triple 1.440415

double 1.813472

surface 2.497409

total\_chairs 7.611399

Runs 41.188482

TerrainParks 2.434783

LongestRun\_mi 1.293122

SkiableTerrain\_ac 448.785340

Snow Making\_ac 129.601190

daysOpenLastYear 110.100629

yearsOpen 56.559585

averageSnowfall 162.310160

projectedDaysOpen 115.920245

NightSkiing\_ac 86.384615

resorts\_per\_state 16.264249

resorts\_per\_100kcapita 0.424802

resorts\_per\_100ksq\_mile 40.957785

resort\_skiable\_area\_ac\_state\_ratio 0.097205

resort\_days\_open\_state\_ratio 0.126014

resort\_terrain\_park\_state\_ratio 0.116022

resort\_night\_skiing\_state\_ratio 0.155024

total\_chairs\_runs\_ratio 0.271441

total\_chairs\_skiable\_ratio 0.070483

fastQuads\_runs\_ratio 0.010401

fastQuads\_skiable\_ratio 0.001633

dtype: float64

By eye, you can immediately tell that your replacement values are much higher than those from using the median.

##### 4.8.1.2.2 Apply the imputation to both train and test splits



X\_tr = X\_train.fillna(X\_defaults\_mean)

X\_te = X\_test.fillna(X\_defaults\_mean)

##### 4.8.1.2.3 Scale the data



scaler = StandardScaler()

scaler.fit(X\_tr)

X\_tr\_scaled = scaler.transform(X\_tr)

X\_te\_scaled = scaler.transform(X\_te)

##### 4.8.1.2.4 Train the model on the train split



lm = LinearRegression().fit(X\_tr\_scaled, y\_train)

##### 4.8.1.2.5 Make predictions using the model on both train and test splits



y\_tr\_pred = lm.predict(X\_tr\_scaled)

y\_te\_pred = lm.predict(X\_te\_scaled)

##### 4.8.1.2.6 Assess model performance



r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_test, y\_te\_pred)

(0.8170154093990025, 0.7163814716959958)



mean\_absolute\_error(y\_train, y\_tr\_pred), mean\_absolute\_error(y\_test, y\_te\_pred)

(8.536884040670977, 9.416375625789277)



mean\_squared\_error(y\_train, y\_tr\_pred), mean\_squared\_error(y\_test, y\_te\_pred)

(112.37695054778276, 164.39269309524374)

These results don't seem very different to when you used the median for imputing missing values. Perhaps it doesn't make much difference here. Maybe your overtraining dominates. Maybe other feature transformations, such as taking the log, would help. You could try with just a subset of features rather than using all of them as inputs.

To perform the median/mean comparison, you copied and pasted a lot of code just to change the function for imputing missing values. It would make more sense to write a function that performed the sequence of steps:

1. impute missing values
2. scale the features
3. train a model
4. calculate model performance

But these are common steps and sklearn provides something much better than writing custom functions.

### 4.8.2 Pipelines

One of the most important and useful components of sklearn is the [pipeline](https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html). In place of panda's fillna DataFrame method, there is sklearn's SimpleImputer. Remember the first linear model above performed the steps:

1. replace missing values with the median for each feature
2. scale the data to zero mean and unit variance
3. train a linear regression model

and all these steps were trained on the train split and then applied to the test split for assessment.

The pipeline below defines exactly those same steps. Crucially, the resultant Pipeline object has a fit() method and a predict() method, just like the LinearRegression() object itself. Just as you might create a linear regression model and train it with .fit() and predict with .predict(), you can wrap the entire process of imputing and feature scaling and regression in a single object you can train with .fit() and predict with .predict(). And that's basically a pipeline: a model on steroids.

#### 4.8.2.1 Define the pipeline



pipe = make\_pipeline(

SimpleImputer(strategy='median'),

StandardScaler(),

LinearRegression()

)



type(pipe)

sklearn.pipeline.Pipeline



hasattr(pipe, 'fit'), hasattr(pipe, 'predict')

(True, True)

#### 4.8.2.2 Fit the pipeline

Here, a single call to the pipeline's fit() method combines the steps of learning the imputation (determining what values to use to fill the missing ones), the scaling (determining the mean to subtract and the variance to divide by), and then training the model. It does this all in the one call with the training data as arguments.



#Code task 15#

#Call the pipe's `fit()` method with `X\_train` and `y\_train` as arguments

pipe.fit(X\_train, y\_train)

Pipeline(steps=[('simpleimputer', SimpleImputer(strategy='median')),

('standardscaler', StandardScaler()),

('linearregression', LinearRegression())])

#### 4.8.2.3 Make predictions on the train and test sets



y\_tr\_pred = pipe.predict(X\_train)

y\_te\_pred = pipe.predict(X\_test)

#### 4.8.2.4 Assess performance



r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_test, y\_te\_pred)

(0.8177988515690604, 0.7209725843435146)

And compare with your earlier (non-pipeline) result:



median\_r2

(0.8177988515690604, 0.7209725843435146)



mean\_absolute\_error(y\_train, y\_tr\_pred), mean\_absolute\_error(y\_test, y\_te\_pred)

(8.547850301825429, 9.407020118581318)



#Compare with your earlier result:



median\_mae

(8.547850301825429, 9.407020118581318)



mean\_squared\_error(y\_train, y\_tr\_pred), mean\_squared\_error(y\_test, y\_te\_pred)

(111.89581253658478, 161.73156451192264)

Compare with your earlier result:



median\_mse

(111.89581253658478, 161.73156451192264)

These results confirm the pipeline is doing exactly what's expected, and results are identical to your earlier steps. This allows you to move faster but with confidence.

## 4.9 Refining The Linear Model

You suspected the model was overfitting. This is no real surprise given the number of features you blindly used. It's likely a judicious subset of features would generalize better. sklearn has a number of feature selection functions available. The one you'll use here is SelectKBest which, as you might guess, selects the k best features. You can read about SelectKBest [here](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html#sklearn.feature_selection.SelectKBest). f\_regression is just the [score function](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_regression.html#sklearn.feature_selection.f_regression) you're using because you're performing regression. It's important to choose an appropriate one for your machine learning task.

### 4.9.1 Define the pipeline

Redefine your pipeline to include this feature selection step:



#Code task 16#

#Add `SelectKBest` as a step in the pipeline between `StandardScaler()` and `LinearRegression()`

#Don't forget to tell it to use `f\_regression` as its score function

pipe = make\_pipeline(

SimpleImputer(strategy='median'),

StandardScaler(),

SelectKBest(f\_regression),

LinearRegression()

)

### 4.9.2 Fit the pipeline



pipe.fit(X\_train, y\_train)

Pipeline(steps=[('simpleimputer', SimpleImputer(strategy='median')),

('standardscaler', StandardScaler()),

('selectkbest',

SelectKBest(score\_func=<function f\_regression at 0x0000023DD91A7A60>)),

('linearregression', LinearRegression())])

### 4.9.3 Assess performance on the train and test set



y\_tr\_pred = pipe.predict(X\_train)

y\_te\_pred = pipe.predict(X\_test)



r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_test, y\_te\_pred)

(0.7674914326052744, 0.6259877354190837)



mean\_absolute\_error(y\_train, y\_tr\_pred), mean\_absolute\_error(y\_test, y\_te\_pred)

(9.501495079727484, 11.201830190332052)

This has made things worse! Clearly selecting a subset of features has an impact on performance. SelectKBest defaults to k=10. You've just seen that 10 is worse than using all features. What is the best k? You could create a new pipeline with a different value of k:

### 4.9.4 Define a new pipeline to select a different number of features



#Code task 17#

#Modify the `SelectKBest` step to use a value of 15 for k

pipe15 = make\_pipeline(

SimpleImputer(strategy='median'),

StandardScaler(),

SelectKBest(f\_regression, k=15),

LinearRegression()

)

### 4.9.5 Fit the pipeline



pipe15.fit(X\_train, y\_train)

Pipeline(steps=[('simpleimputer', SimpleImputer(strategy='median')),

('standardscaler', StandardScaler()),

('selectkbest',

SelectKBest(k=15,

score\_func=<function f\_regression at 0x0000023DD91A7A60>)),

('linearregression', LinearRegression())])

### 4.9.6 Assess performance on train and test data



y\_tr\_pred = pipe15.predict(X\_train)

y\_te\_pred = pipe15.predict(X\_test)



r2\_score(y\_train, y\_tr\_pred), r2\_score(y\_test, y\_te\_pred)

(0.7924096060483825, 0.6376199973170795)



mean\_absolute\_error(y\_train, y\_tr\_pred), mean\_absolute\_error(y\_test, y\_te\_pred)

(9.211767769307116, 10.488246867294357)

You could keep going, trying different values of k, training a model, measuring performance on the test set, and then picking the model with the best test set performance. There's a fundamental problem with this approach: you're tuning the model to the arbitrary test set! If you continue this way you'll end up with a model works well on the particular quirks of our test set but fails to generalize to new data. The whole point of keeping a test set is for it to be a set of that new data, to check how well our model might perform on data it hasn't seen.

The way around this is a technique called cross-validation. You partition the training set into k folds, train our model on k-1 of those folds, and calculate performance on the fold not used in training. This procedure then cycles through k times with a different fold held back each time. Thus you end up building k models on k sets of data with k estimates of how the model performs on unseen data but without having to touch the test set.

### 4.9.7 Assessing performance using cross-validation



cv\_results = cross\_validate(pipe15, X\_train, y\_train, cv=5)



cv\_scores = cv\_results['test\_score']

cv\_scores

array([0.63760862, 0.72831381, 0.74443537, 0.5487915 , 0.50441472])

Without using the same random state for initializing the CV folds, your actual numbers will be different.



np.mean(cv\_scores), np.std(cv\_scores)

(0.6327128053007864, 0.0950248784987771)

These results highlight that assessing model performance in inherently open to variability. You'll get different results depending on the quirks of which points are in which fold. An advantage of this is that you can also obtain an estimate of the variability, or uncertainty, in your performance estimate.



np.round((np.mean(cv\_scores) - 2 \* np.std(cv\_scores), np.mean(cv\_scores) + 2 \* np.std(cv\_scores)), 2)

array([0.44, 0.82])

### 4.9.8 Hyperparameter search using GridSearchCV

Pulling the above together, we have:

* a pipeline that
  + imputes missing values
  + scales the data
  + selects the k best features
  + trains a linear regression model
* a technique (cross-validation) for estimating model performance

Now you want to use cross-validation for multiple values of k and use cross-validation to pick the value of k that gives the best performance. make\_pipeline automatically names each step as the lowercase name of the step and the parameters of the step are then accessed by appending a double underscore followed by the parameter name. You know the name of the step will be 'selectkbest' and you know the parameter is 'k'.

You can also list the names of all the parameters in a pipeline like this:



#Code task 18#

#Call `pipe`'s `get\_params()` method to get a dict of available parameters and print their names

#using dict's `keys()` method

pipe.get\_params().keys()

dict\_keys(['memory', 'steps', 'verbose', 'simpleimputer', 'standardscaler', 'selectkbest', 'linearregression', 'simpleimputer\_\_add\_indicator', 'simpleimputer\_\_copy', 'simpleimputer\_\_fill\_value', 'simpleimputer\_\_missing\_values', 'simpleimputer\_\_strategy', 'simpleimputer\_\_verbose', 'standardscaler\_\_copy', 'standardscaler\_\_with\_mean', 'standardscaler\_\_with\_std', 'selectkbest\_\_k', 'selectkbest\_\_score\_func', 'linearregression\_\_copy\_X', 'linearregression\_\_fit\_intercept', 'linearregression\_\_n\_jobs', 'linearregression\_\_normalize', 'linearregression\_\_positive'])

The above can be particularly useful as your pipelines becomes more complex (you can even nest pipelines within pipelines).



k = [k+1 for k in range(len(X\_train.columns))]

grid\_params = {'selectkbest\_\_k': k}

Now you have a range of k to investigate. Is 1 feature best? 2? 3? 4? All of them? You could write a for loop and iterate over each possible value, doing all the housekeeping oyurselves to track the best value of k. But this is a common task so there's a built in function in sklearn. This is [GridSearchCV](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html). This takes the pipeline object, in fact it takes anything with a .fit() and .predict() method. In simple cases with no feature selection or imputation or feature scaling etc. you may see the classifier or regressor object itself directly passed into GridSearchCV. The other key input is the parameters and values to search over. Optional parameters include the cross-validation strategy and number of CPUs to use.



lr\_grid\_cv = GridSearchCV(pipe, param\_grid=grid\_params, cv=5, n\_jobs=-1)



lr\_grid\_cv.fit(X\_train, y\_train)

GridSearchCV(cv=5,

estimator=Pipeline(steps=[('simpleimputer',

SimpleImputer(strategy='median')),

('standardscaler', StandardScaler()),

('selectkbest',

SelectKBest(score\_func=<function f\_regression at 0x0000023DD91A7A60>)),

('linearregression',

LinearRegression())]),

n\_jobs=-1,

param\_grid={'selectkbest\_\_k': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,

12, 13, 14, 15, 16, 17, 18, 19, 20,

21, 22, 23, 24, 25, 26, 27, 28, 29,

30, ...]})



score\_mean = lr\_grid\_cv.cv\_results\_['mean\_test\_score']

score\_std = lr\_grid\_cv.cv\_results\_['std\_test\_score']

cv\_k = [k for k in lr\_grid\_cv.cv\_results\_['param\_selectkbest\_\_k']]



#Code task 19#

#Print the `best\_params\_` attribute of `lr\_grid\_cv`

lr\_grid\_cv.best\_params\_

{'selectkbest\_\_k': 8}



#Code task 20#

#Assign the value of k from the above dict of `best\_params\_` and assign it to `best\_k`

best\_k = lr\_grid\_cv.best\_params\_['selectkbest\_\_k']

plt.subplots(figsize=(10, 5))

plt.errorbar(cv\_k, score\_mean, yerr=score\_std)

plt.axvline(x=best\_k, c='r', ls='--', alpha=.5)

plt.xlabel('k')

plt.ylabel('CV score (r-squared)')

plt.title('Pipeline mean CV score (error bars +/- 1sd)');

Chart

Description automatically generated

The above suggests a good value for k is 8. There was an initial rapid increase with k, followed by a slow decline. Also noticeable is the variance of the results greatly increase above k=8. As you increasingly overfit, expect greater swings in performance as different points move in and out of the train/test folds.

Which features were most useful? Step into your best model, shown below. Starting with the fitted grid search object, you get the best estimator, then the named step 'selectkbest', for which you can its get\_support() method for a logical mask of the features selected.



selected = lr\_grid\_cv.best\_estimator\_.named\_steps.selectkbest.get\_support()

Similarly, instead of using the 'selectkbest' named step, you can access the named step for the linear regression model and, from that, grab the model coefficients via its coef\_ attribute:



#Code task 21#

#Get the linear model coefficients from the `coef\_` attribute and store in `coefs`,

#get the matching feature names from the column names of the dataframe,

#and display the results as a pandas Series with `coefs` as the values and `features` as the index,

#sorting the values in descending order

coefs = lr\_grid\_cv.best\_estimator\_.named\_steps.linearregression.coef\_

features = X\_train.columns[selected]

pd.Series(coefs, index=features).sort\_values(ascending=False)

vertical\_drop 10.767857

Snow Making\_ac 6.290074

total\_chairs 5.794156

fastQuads 5.745626

Runs 5.370555

LongestRun\_mi 0.181814

trams -4.142024

SkiableTerrain\_ac -5.249780

dtype: float64

These results suggest that vertical drop is your biggest positive feature. This makes intuitive sense and is consistent with what you saw during the EDA work. Also, you see the area covered by snow making equipment is a strong positive as well. People like guaranteed skiing! The skiable terrain area is negatively associated with ticket price! This seems odd. People will pay less for larger resorts? There could be all manner of reasons for this. It could be an effect whereby larger resorts can host more visitors at any one time and so can charge less per ticket. As has been mentioned previously, the data are missing information about visitor numbers. Bear in mind, the coefficient for skiable terrain is negative for this model. For example, if you kept the total number of chairs and fastQuads constant, but increased the skiable terrain extent, you might imagine the resort is worse off because the chairlift capacity is stretched thinner.

## 4.10 Random Forest Model

A model that can work very well in a lot of cases is the random forest. For regression, this is provided by sklearn's RandomForestRegressor class.

Time to stop the bad practice of repeatedly checking performance on the test split. Instead, go straight from defining the pipeline to assessing performance using cross-validation. cross\_validate will perform the fitting as part of the process. This uses the default settings for the random forest so you'll then proceed to investigate some different hyperparameters.

### 4.10.1 Define the pipeline



#Code task 22#

#Define a pipeline comprising the steps:

#SimpleImputer() with a strategy of 'median'

#StandardScaler(),

#and then RandomForestRegressor() with a random state of 47

RF\_pipe = make\_pipeline(

SimpleImputer(strategy='median'),

StandardScaler(),

RandomForestRegressor(random\_state=47)

)

### 4.10.2 Fit and assess performance using cross-validation



#Code task 23#

#Call `cross\_validate` to estimate the pipeline's performance.

#Pass it the random forest pipe object, `X\_train` and `y\_train`,

#and get it to use 5-fold cross-validation

rf\_default\_cv\_results = cross\_validate(RF\_pipe, X\_train, y\_train, cv=5)



rf\_cv\_scores = rf\_default\_cv\_results['test\_score']

rf\_cv\_scores

array([0.69249204, 0.78061953, 0.77546915, 0.62190924, 0.61742339])



np.mean(rf\_cv\_scores), np.std(rf\_cv\_scores)

(0.6975826707112506, 0.07090742940774528)

### 4.10.3 Hyperparameter search using GridSearchCV

Random forest has a number of hyperparameters that can be explored, however here you'll limit yourselves to exploring some different values for the number of trees. You'll try it with and without feature scaling, and try both the mean and median as strategies for imputing missing values.



n\_est = [int(n) for n in np.logspace(start=1, stop=3, num=20)]

grid\_params = {

'randomforestregressor\_\_n\_estimators': n\_est,

'standardscaler': [StandardScaler(), None],

'simpleimputer\_\_strategy': ['mean', 'median']

}

grid\_params

{'randomforestregressor\_\_n\_estimators': [10,

12,

16,

20,

26,

33,

42,

54,

69,

88,

112,

143,

183,

233,

297,

379,

483,

615,

784,

1000],

'standardscaler': [StandardScaler(), None],

'simpleimputer\_\_strategy': ['mean', 'median']}



#Code task 24#

#Call `GridSearchCV` with the random forest pipeline, passing in the above `grid\_params`

#dict for parameters to evaluate, 5-fold cross-validation, and all available CPU cores (if desired)

rf\_grid\_cv = GridSearchCV(RF\_pipe, param\_grid=grid\_params, cv=5, n\_jobs=-1)



#Code task 25#

#Now call the `GridSearchCV`'s `fit()` method with `X\_train` and `y\_train` as arguments

#to actually start the grid search. This may take a minute or two.

rf\_grid\_cv.fit(X\_train, y\_train)

GridSearchCV(cv=5,

estimator=Pipeline(steps=[('simpleimputer',

SimpleImputer(strategy='median')),

('standardscaler', StandardScaler()),

('randomforestregressor',

RandomForestRegressor(random\_state=47))]),

n\_jobs=-1,

param\_grid={'randomforestregressor\_\_n\_estimators': [10, 12, 16, 20,

26, 33, 42, 54,

69, 88, 112,

143, 183, 233,

297, 379, 483,

615, 784,

1000],

'simpleimputer\_\_strategy': ['mean', 'median'],

'standardscaler': [StandardScaler(), None]})



#Code task 26#

#Print the best params (`best\_params\_` attribute) from the grid search

rf\_grid\_cv.best\_params\_

{'randomforestregressor\_\_n\_estimators': 69,

'simpleimputer\_\_strategy': 'median',

'standardscaler': None}

It looks like imputing with the median helps, but scaling the features doesn't.



rf\_best\_cv\_results = cross\_validate(rf\_grid\_cv.best\_estimator\_, X\_train, y\_train, cv=5)

rf\_best\_scores = rf\_best\_cv\_results['test\_score']

rf\_best\_scores

array([0.6951357 , 0.79430697, 0.77170917, 0.62254707, 0.66499334])



np.mean(rf\_best\_scores), np.std(rf\_best\_scores)

(0.7097384501425082, 0.06451341966873386)

You've marginally improved upon the default CV results. Random forest has many more hyperparameters you could tune, but we won't dive into that here.



#Code task 27#

#Plot a barplot of the random forest's feature importances,

#assigning the `feature\_importances\_` attribute of

#`rf\_grid\_cv.best\_estimator\_.named\_steps.randomforestregressor` to the name `imps` to then

#create a pandas Series object of the feature importances, with the index given by the

#training data column names, sorting the values in descending order

plt.subplots(figsize=(10, 5))

imps = rf\_grid\_cv.best\_estimator\_.named\_steps.randomforestregressor.feature\_importances\_

rf\_feat\_imps = pd.Series(imps, index=X\_train.columns).sort\_values(ascending=False)

rf\_feat\_imps.plot(kind='bar')

plt.xlabel('features')

plt.ylabel('importance')

plt.title('Best random forest regressor feature importances');

Chart, histogram

Description automatically generated

Encouragingly, the dominant top four features are in common with your linear model:

* fastQuads
* Runs
* Snow Making\_ac
* vertical\_drop

## 4.11 Final Model Selection

Time to select your final model to use for further business modeling! It would be good to revisit the above model selection; there is undoubtedly more that could be done to explore possible hyperparameters. It would also be worthwhile to investigate removing the least useful features. Gathering or calculating, and storing, features adds business cost and dependencies, so if features genuinely are not needed they should be removed. Building a simpler model with fewer features can also have the advantage of being easier to sell (and/or explain) to stakeholders. Certainly there seem to be four strong features here and so a model using only those would probably work well. However, you want to explore some different scenarios where other features vary so keep the fuller model for now. The business is waiting for this model and you have something that you have confidence in to be much better than guessing with the average price.

Or, rather, you have two "somethings". You built a best linear model and a best random forest model. You need to finally choose between them. You can calculate the mean absolute error using cross-validation. Although cross-validate defaults to the *𝑅*2

[metric for scoring](https://scikit-learn.org/stable/modules/model_evaluation.html#scoring) regression, you can specify the mean absolute error as an alternative via the scoring parameter.

### 4.11.1 Linear regression model performance



# 'neg\_mean\_absolute\_error' uses the (negative of) the mean absolute error

lr\_neg\_mae = cross\_validate(lr\_grid\_cv.best\_estimator\_, X\_train, y\_train,

scoring='neg\_mean\_absolute\_error', cv=5, n\_jobs=-1)



lr\_mae\_mean = np.mean(-1 \* lr\_neg\_mae['test\_score'])

lr\_mae\_std = np.std(-1 \* lr\_neg\_mae['test\_score'])

lr\_mae\_mean, lr\_mae\_std

(10.499032338015294, 1.6220608976799664)



mean\_absolute\_error(y\_test, lr\_grid\_cv.best\_estimator\_.predict(X\_test))

11.793465668669327

### 4.11.2 Random forest regression model performance



rf\_neg\_mae = cross\_validate(rf\_grid\_cv.best\_estimator\_, X\_train, y\_train,

scoring='neg\_mean\_absolute\_error', cv=5, n\_jobs=-1)



rf\_mae\_mean = np.mean(-1 \* rf\_neg\_mae['test\_score'])

rf\_mae\_std = np.std(-1 \* rf\_neg\_mae['test\_score'])

rf\_mae\_mean, rf\_mae\_std

(9.644639167595688, 1.3528565172191818)



mean\_absolute\_error(y\_test, rf\_grid\_cv.best\_estimator\_.predict(X\_test))

9.537730050637332

### 4.11.3 Conclusion

The random forest model has a lower cross-validation mean absolute error by almost $1. It also exhibits less variability. Verifying performance on the test set produces performance consistent with the cross-validation results.

## 4.12 Data quantity assessment

Finally, you need to advise the business whether it needs to undertake further data collection. Would more data be useful? We're often led to believe more data is always good, but gathering data invariably has a cost associated with it. Assess this trade off by seeing how performance varies with differing data set sizes. The learning\_curve function does this conveniently.



fractions = [.2, .25, .3, .35, .4, .45, .5, .6, .75, .8, 1.0]

train\_size, train\_scores, test\_scores = learning\_curve(pipe, X\_train, y\_train, train\_sizes=fractions)

train\_scores\_mean = np.mean(train\_scores, axis=1)

train\_scores\_std = np.std(train\_scores, axis=1)

test\_scores\_mean = np.mean(test\_scores, axis=1)

test\_scores\_std = np.std(test\_scores, axis=1)



plt.subplots(figsize=(10, 5))

plt.errorbar(train\_size, test\_scores\_mean, yerr=test\_scores\_std)

plt.xlabel('Training set size')

plt.ylabel('CV scores')

plt.title('Cross-validation score as training set size increases');

Chart, box and whisker chart

Description automatically generated

This shows that you seem to have plenty of data. There's an initial rapid improvement in model scores as one would expect, but it's essentially levelled off by around a sample size of 40-50.

## 4.13 Save best model object from pipeline



​



#Code task 28#

#This may not be "production grade ML deployment" practice, but adding some basic

#information to your saved models can save your bacon in development.

#Just what version model have you just loaded to reuse? What version of `sklearn`

#created it? When did you make it?

#Assign the pandas version number (`pd.\_\_version\_\_`) to the `pandas\_version` attribute,

#the numpy version (`np.\_\_version\_\_`) to the `numpy\_version` attribute,

#the sklearn version (`sklearn\_version`) to the `sklearn\_version` attribute,

#and the current datetime (`datetime.datetime.now()`) to the `build\_datetime` attribute

#Let's call this model version '1.0'

best\_model = rf\_grid\_cv.best\_estimator\_

best\_model.version = '1.0'

best\_model.pandas\_version = 'pd.\_version\_'

best\_model.numpy\_version = 'np.\_version\_'

best\_model.sklearn\_version = 'sklearn\_version'

best\_model.X\_columns = [col for col in X\_train.columns]

best\_model.build\_datetime = datetime.datetime.now()

Type Markdown and LaTeX: *𝛼*2



# save the model

​

modelpath = '../models'

save\_file(best\_model, 'ski\_resort\_pricing\_model.pkl', modelpath)

A file already exists with this name.

Do you want to overwrite? (Y/N)Y

Writing file. "../models\ski\_resort\_pricing\_model.pkl"

## 4.14 Summary

**Q: 1** Write a summary of the work in this notebook. Capture the fact that you gained a baseline idea of performance by simply taking the average price and how well that did. Then highlight that you built a linear model and the features that found. Comment on the estimate of its performance from cross-validation and whether its performance on the test split was consistent with this estimate. Also highlight that a random forest regressor was tried, what preprocessing steps were found to be best, and again what its estimated performance via cross-validation was and whether its performance on the test set was consistent with that. State which model you have decided to use going forwards and why. This summary should provide a quick overview for someone wanting to know quickly why the given model was chosen for the next part of the business problem to help guide important business decisions.

**A: 1** The first model is the baseline performance to compare with the other models. Here, the average price(mean) is used as a predictor. The training data and the dummy regressor fit training means were found to closely match. The calculated average price is 63.81. This seems to indicate a good fit but it would just be an average of the values and it may not be a good predictor.

Next metric will tell us how good the average is as a predictor. The R^2 (coefficient of determination) was used. Results of R^2 of training set of zero and R^2 of the test set as -0.0031, indicating a very poor fit for prediction. Generally, the performance of the test set is expected to be slightly worse than on the training set.

Next metric, will summarize the difference between predicted and actual values using the mean absolute error and mean squared error. MAE (training set)=17.923 and MAE (test set)=19.236. This means MAE on average expected to be off by around $19 if you guessed ticket price based on an average of known values.

MSE (training set)=614.133, MSE (test set)=581.436. Here, we have a slightly better MSE on the test set than we did on the training set.

Next, imputing missing values using median for both training and test sets then scaled each feature to zero mean and unit variance. This was done because of the skew of many of the predictor feature distributions.

The average was used to estimate ticket prices. Result: R^2 of zero for the training and test set. This tells us that the model is over fit, and not a good value for predictive purposes.

Using a linear regression fit model, the median for imputing missing value, explained over 80% of the variance on the training set and over 70% on the test set. The noted lower value for the test set suggests possible overfitting on the training data. We obtained R^2 values for training of 0.817 and for test of 0.720.

A second linear regression fit model using the mean for imputing missing values, yielded R^2 for training and test of 0.817 and 0.716, respectively.

The results of R^2, MAE, MSE for both median and mean do not seem very different, showing it may not make much difference and we may have overtraining in the models. The data suggests trying a subset of features rather than using all of them as inputs.

The linear model needs to be refined because the model was suspected to be overfitting. So, selectKbest was used to select the best features performing f-regression. Results were noted to be worse. selecting a subset of features has an impact on performance. Using the default k of 10 means using 10 features is worse than using all features. Even if different values for k are used, when measuring the performance on the test set and picking the model with the best test set performance, it shows the model is being tuned to the arbitrary test set. This results in a model that works well on the particular quirks of our test set but fails to generalize to new data. This defeats the purpose of keeping the test set of new data to check how well our model might perform on data it hasn’t seen. So, the next step is cross validation to estimate model performance.

Cross validation was performed to build models on k sets of data with k estimates of how the model performs on unseen data but without having to touch the test set.

The results highlight that assessing model performance is inherently open to variability. This will get different results depending on the quirks of which points are in which fold, The advantage of this is that you can also obtain an estimate of the variability, or uncertainty, in the performance estimate.

The pipeline mean cross validation score showed a good value for k is 8. This showed an initial rapid increase with k followed by a slow decline. Also noticeable was the variance of the results greatly increased above k=8. As you increasingly overfit, greater swings in performance were seen as different points move in and out of the training/test folds.

Results suggest that vertical drop is the biggest positive feature. Which was consistent with what was seen in the EDA work. Another is the area covered by snow making equipment is a positive feature. The skiable terrain area and trams are negatively associated with ticket price. Does this mean that people will pay less for larger resorts? It could be an effect whereby larger resorts can host more visitors at any one time and so can charge less per ticket. This reminds us about the data on missing information about visitor numbers might still be of interest.

The business needs to be advised whether to undertake further data collection. Gathering data has a cost associated to it so a data quantity assessment tool called learning curve function is used to see how performance varies with differing data set size.

The results from the learning curve function showed that as training set size increases the cross validation score increases. There is an initial rapid improvement in model scores as one would expect, but it’s essentially leveled off by around a sample size of 40-50 showing that there seems to be plenty of data.

Next, the use of random forest regressor model to estimate pipeline performance using cross validation. The final results for the top dominant four features are consistent with the linear model are fast quads, runs, snow making area, vertical drop.

In choosing the model, building a simpler model with fewer features can be advantageous making it easier to sell and explain to stakeholders. We want to build the best linear model and best random forest model, to compare. The final linear regression model performance yielded an MAE of 11.793, with the random forest regression model performance with an MAE of 9.537. With this information the random forest regression model was chosen.

The random forest model has a lower cross validation mean absolute error by almost $1. It also exhibits less variability. Verifying performance on the test set produces performance consistent with the cross validation results. From these results, the random forest regression model was chosen for the business problem to help guide important business decisions.