Machine Learning with Python: Foundations

- Computers receive 2 things
  - An input or data
  - A set of instructions on what to do
- ML is where computers don't get an explicit set of instructions
  - We give an input and expected output
  - Computer figures out the instructions
- Focus of ML is to predict
- 6 Phases of ML
  1. Data Collection
  2. Data Exploration
  3. Data Preparation
  4. Model
  5. Evaluate Model
  6. Actionable Insight

Data Collection

Considerations

- Objective is to identify and gather data to be used
- There are 5 key considerations when it comes to data
- The first is *accuracy*
  - Accurate data is needed for a good model
  - For supervised learning, this is especially crucial
    - Bad data = bad predictions
  - Labels for data can come from 2 sources
    - Event-based (it's a label based on a fact that happened)
    - Assigned (usually by experts on the topic)
  - It's important to have a way to validate data after it's collected and labelled
- Second is *relevance*
  - Data needs to be relevant in explaining why an input resulted in a certain output
    - Ex: shoe sizes have nothing to do with loan payment rates
- Third is *quantity*
  - Generally speaking, most algorithms will need a lot of data for meaningful results
- Fourth is *variability*
  - Data needs to be diverse
  - Algorithm gains a broader view of the domain
- Fifth is *ethics*
  - Concerns include bias, security, privacy, and consent
  - Biased data leads to biased predictions

        - Can be unintentional or implicit from human ethics
        - Need to minimize as much as possible

Import Data in Python

```
import pandas
```

- pandas is the go-to package when it comes to data analysis
- Lots of great functions that makes it easy to use

Basic Data Structures

```
# a 1d list is called a 'series'
fruits = ["apple", "banana", "orange"]
series = pandas.Series(fruits)

# a 2d list is a 'dataframe'
myDict = {
    "letters": ['a','b','c'],
    "numbers": [1,2,3],
    "decimals": [1.1, 2.2, 3.3]
}
df = pandas.DataFrame(myDict)

# custom column names
labels = ["label1", "label2", "label3"]
df2 = pandas.DataFrame(myDict, labels)
```

- Data structures are heterogenous
    - Can store different kinds of data
    - 1D is a *series* and 2D is a *dataframe*
- Dataframes can either infer the column labels or you can pass in a custom list for the labels

Reading from Files

```
csv = pandas.read_csv("path/to/file.csv")

# if an Excel file has multiple sheets, it'll read the first one by default
excel = pandas.read_excel("path/to/file.xlsx")

# specify sheet for Excel
excel2 = pandas.read_excel("path/to/file.xlsx", sheet_name="sheet2")
```

Data Exploration

Describing Data

- The idea of data exploration is to be able to describe our data
  - What it's about, how much data, and what's the quality
- A single "row" of data is called an instance
  - Also called a *record* or *observation*
  - This is a single example of the general concept behind the data collection
- Features describe important characteristics of the instance
  - Categorical: holds discrete data, limited to a set of possible values
  - Continuous: usually integers or real numbers, infinite possibilities
- Sometimes we want to predict features of a dataset
- Dimensionality is the number of features in a dataset
  - More dimensions means more details
  - Also means higher computational complexity
- Sparsity and density describe how much data we have
  - Complementing concepts
  - Ex: missing 20% data = 20% sparse, 80% dense

Describing Data in Python

Basics

```
data = pandas.read_csv("dataFile.csv")
data.info()
print(data.head())
```

- The `info()` command will display a bunch of useful info describing our dataset
  - Dimensions
  - Types of data
- `head()` gives a sneak-peek of what our data actually looks like
  - Returns first 5 rows

Aggregations

```
print(data[["ColumnName"]].describe())
```

- This allows us to get info regarding a specific attribute
- It'll return 4 basic info for non-numbers
  - `count`: number of (nonempty) entries
  - `unique`: number of unique entries

    - `top`: the most frequently occurring entry
    - `freq`: how often it occurs
- In columns that are statistical, it'll return
    - `count`: number of entries
    - `mean`
    - `std`
    - `25%`
    - `50%` (aka median)
    - `75%`
    - `max`

Getting Value Counts

```
# gives a count of each unique value
data[["ColumnName"]].value_counts()

# gives percent of each unique value
data[["ColumnName"]].value_counts(normalize=True)
```

Other Functions

```
# get the mean of values in a column
data[["ColumnName"]].mean()

# group instances, and then get the mean values of each group
data.groupby("ColumnName1")[["ColumnName2"]].mean()

# sort based on a column (default is ascending)
data.groupby("ColumnName1")[["ColumnName2"]].mean().sort_values(by="ColumnName3")
```

Multiple Aggregations

```
# 'agg' gives us multiple aggregations
data.groupby("ColumnName1")[["ColumnName2"]].agg(["mean", "median", "max"])
```

Data Visualizations

- It's sometimes easier to show patterns with visuals
- Comparison visualizations show the difference between 2 or more items
    - Ex: a box plot

- Relationship visualizations shows how 2+ variables can affect each other
  - Ex: line charts and scatter plots
- Distribution visualizations shows stat distribution of a single feature
  - Ex: histograms
- Composition visualizations show the inner makeup of data
  - Ex: stacked bar charts, pie charts

Data Visualization in Python

```
import pandas
import matplotlib.pyplot as plt

vehicles = pandas.read_csv("vehicles.csv")
```

- `matplotlib` is a popular package for generating data visualizations
- Note that in order to actually display the plot, we need to run `plt.show()` at the end

Scatter

```
vehicles.plot.scatter(x="citympg",y="co2emissions")
plt.show()
```

Histogram

```
vehicles[["co2emissions"]].plot.hist()
plt.show()
```

Box

```
pivot = vehicles.pivot(columns="drive", values="co2emissions")
pivot.boxplot(figsize=(10,6))
plt.show()
```

- In order to use a box plot, we need to first create a pivot table
  - X-axis is column values, Y-axis is cell values

Stacked Bar Chart

```
pivot = vehicles.groupby("year")["drive"].value_counts().unstack()
pivot.plot.bar(stacked=True, figsize=(10,6))
plt.show()
```

- We need to first create a pivot
    - We group by year
    - Then aggregate by drive

Data Preparation

Common Data Quality Issues

- The process of ensuring the data is suitable for our ML model
- One of the most common data issues is *missing data*
    - Usually a few missing features in a record here and there
    - Can result from human error, bias, or lack of reliable input
    - Changes in data collection methods can also result in missing data
- Several ways to resolve this
    - Remove records with missing data
    - Use holders to represent missing data (like `N/A` or `-1`)
    - We can also try to use *imputation*, which is a process to reasonably guess what those values are
        - Ex: replace all missing values with the median of non-missing values
- Another issue is *outliers*
    - Features that are unusual or very different than others
    - To fix outliers, you need to first understand what the outliers convey
- One valid reason for outliers is *class imbalance*
    - This is when the real world distribution of values is not uniform
    - More instances of a class label than others
    - Can lead to misleading predictions if not properly handled
- Ways to resolve
    - Undersample majority classes

Resolving Missing Data in Python

Check for Missing Values

```
import pandas

students = pandas.read_excel("students.xlsx")
mask = students['State'].isnull()

# display 'True' if attribute is missing, 'False' otherwise
print(mask)
```

```
# only display rows with missing data
print(students[mask])
```

Remove Missing Values

```
# remove all rows with any missing values
students.dropna()
```

- This is a rather extreme approach
- We usually want to remove rows that have missing values *in a particular column*

```
# only drop rows if both State and Zip are missing
students.dropna(subset=["State","Zip"], how="all")

# drop columns with any missing values
students.dropna(axis=1)

# specifies how many values need to be missing before it's removed
students.dropna(axis=1, thresh=10)
```

Resolve Missing Values

```
# replace all missing values with a constant value
students.fillna({'Gender':'Female'})

# replace all missing values with a value from a function
students.fillna({'Age':students['Age'].median()})

# change value of specific cell
mask = (studnets['City'] == 'Granger') & (students['State'] == 'IN')
students.loc[mask, 'Zip'] = 46530
```

Normalizing Data

- Goal of *normalization* (also called *standardization*) is to ensure all data share a common property
    - Typically involves scaling data to fit within a range
    - Reduces complexity and improves interpretability
- *Z-Score* Normalization is one approach
    - Ensures the data has a mean of 0 and std of 1

- $v' = \frac{v-\bar{F}}{\sigma_F}$
  - $\bar{F}$ is mean
  - σ_F is std
  - F is feature
  - v and v' are the old and new values
- Ex: suppose we have the values $12000, 24000, 30000, 40000, 98000$
  - Mean is 40800, std is 33544
  - v=40000: $v' = \frac{40000-40800}{33544} = -0.024$
  - Final normalized values are $-0.859, -0.500, -0.322, -0.024, -1.705$
- Mean-max normalization gives our data a range between 2 user-defined bounds
  - Typically the bounds are 0 and 1
  - $v' = \frac{v-min_F}{max_F-min_F}(upper_F - lower_F) + lower_F$
  - Ex: bound $12000, 24000, 30000, 40000, 98000$ to 0 and 1
    - v=30000: $v' = \frac{30000-12000}{98000-12000}(1-0) + 0 = 0.209$
    - Final normalized values are $0.000, 0.140, 0.209, 0.326, 1.000$
- Both these approaches are suitable for data with no significant outliers
- Log transformation works for data with lots of outliers
  - $v' = \log v$
  - Base can either be 2 or 10
  - Note that this only works for positive values

Normalize Data in Python

- We use the `scikit-learn` package for data transformations

Min-Max

```
from sklearn.preprocessing import MinMaxScaler

# transform data
c02emissions_mm = MinMaxScaler().fit_transform(vehicles[['co2emissions']])

# convert to pandas dataframe
c02emissions_mm = pandas.DataFrame(co2emissions_mm, columns=['co2emissions'])

# we can then describe and plot it
print(co2emissions_mm.describe())
c02emissions_mm.plot.hist(bins=20, figsize=(10,6))
```

- The histogram for the original and transformed data will look the same
  - What changed is the scale of the X-axis

Z-Score

```
from sklearn.preprocessing import StandardScaler

c02emissions_zm = StandardScaler().fit_transform(vehicles[['co2emissions']])
c02emissions_zm = pandas.DataFrame(c02emissions_zm, columns=['co2emissions'])
print(c02emissions_zm.describe())
c02emissions_zm.plot.hist(bins=20, figsize=(10,6))
```

- Same process, just different Scaler object

Sampling

- Sometimes we need to split our current dataset
  - Current data could be too big
  - We might want to hold on to some of the data for later use
- This process is called sampling
  - The overall dataset is called a population
  - The subset we chose is called the sample
- Sampling methods
  - Random sampling without replacement
  - Random sampling with replacement
    - Also called *bootstrapping*
    - Good for datasets with little data
  - Stratified random sampling
    - Ensures the sample's distribution matches the population's

Sample Data in Python

```
response = 'co2emissions'
y = vehicles[[response]]

predictors = list(vehicles.columns)
predictors.remove(response)
x = vehicles[predictors]
```

- Suppose in `vehicles.csv`, we want `co2emissions` to be the output based on the other variables
  - `co2emissions` is the response
  - The others are the predictors

Split Data using Simple Random Sampling

```
from sklearn.model_selection import train_test_split
```

```
x_train, x_test, y_train, y_test = train_test_split(x, y)
```

- `x_train` holds independent variables of training set
- `y_train` holds dependent variables of training set
- `x_test` holds independent variables of testing set
- `y_test` holds dependent variables of testing set

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.4)
```

- By default, 75% of dataset goes to training and 25% goes to test
- We can change it with `test_size`

Split Data using Stratified Random Sampling

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.4,
stratify=x['drive'])
```

- We use `stratify` to specify which feature (column) to stratify by
- This ensures the dataset's distribution matches the distribution of population more closely

Reducing Data Dimensionality

- Idea is to reduce number of features before modeling
- The "curse of dimensionality" is the idea that increasing features will eventually decrease the performance of the model
    - We need exponentially more data instances per feature
    - Unless we can provide more data instances, our model will decrease
    - There's a limit to how much data was can collect, leading to a limit to the number of features we should have for optimal performance
- Feature selection is one way to reduce dimensionality
    - Identify min number of features needed for a good performance
    - Remove features that have minimal impact on the model performance
- Feature extraction is another method
    - Use math to reduce it
    - Results in completely new features
    - While still reliable, the disadvantage is that the new feature values are harder to interpret as a user

Modeling & Evaluation

Modeling

- The most well known phase of ML
- Objective is to identify the best ML modeling approach to solve the problem
- For supervised learning, there's 2 categories
    - *Classification*: use features to produce a label
    - *Regression*: use features to predict a continuous value
- Lots of different ML techniques can be used for both classification and regression
- There are also some techniques that are specifically for regression problems

Evaluation

- To evaluate a model, we need to run it on data it's never seen before
    - This ensures we have a reliable assessment of the model
- We usually grade the model with an accuracy metric
    - For classification, we use % correct
    - For regression, we use Mean Absolute Error: $MAE = \frac{\sum |Predicted - Actual|}{NumberTestInstances}$
        - This gives us an average margin of error

Building Model in Python

Build Model

```
import pandas
bikes = pandas.read_csv("bikes.csv")

response = "rentals"
y = bikes[[response]]

predictors = list(bikes.columns)
predictors.remove(response)
x = bikes[predictors]

from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y)
```

Train Model

```
from sklearn.linear_model import LinearRegression
model = LinearRegressions().fit(x_train, y_train)
```

- Linear regression assumes there's a linear relationship between each feature and the output

```
# useful variable values
model.intercept_
model.coef_
```

- `intercept_` gives us the intercept for the equation
  - It's a constant that gets added to the end of calculations
- `coef_` lists corresponding weights for each feature in the respective order
- These 2 variables basically give us the equation in which we can make predictions manually

Evaluate Model

```
model.score(x_test, y_test)
```

- This gives us the R-squared value
  - R-square value is also called the *coefficient of determination*
  - Common measurement for linear regression models
  - The close it is to 1, the better it is
- A R-square value of 0.98 means it explains 98% of the variability in response values for test data

```
from sklearn.metrics import mean_absolute_error
y_pred = model.predict(x_test)
mean_absolute_error(y_test, y_pred)
```

- This gives us the MAE, which is an average margin of error