U of M Data Analysis BootCamp 2022
HR Employee Attrition
Capstone Project Group 1
Phil Minor, John Madden, Oliver Dykstra, Conor Hanson

The Dilemma of Employee Attrition

Employee attrition in the workplace is disruptive. It is disruptive to a company's fiscal goals, the quality and timeliness of goods and services supplied, and could potentially tarnish the company's reputation. What's more, it's also incredibly disruptive to employees' own lives. Therefore, avoiding employee attrition is in the best interest of both employers and employees. Drawing on an Human Resources Attrition dataset found on Kaggle, and containing over 4,000 rows of data, we built a machine learning model to predict what factors may or may not correlate with employee attrition. This predictive model is interactive on our full stack Heroku website, along with interactive data visualizations via Tableau and Plotly, a SQLite database which the user can query, and other pertinent documentation.

Initially, our expectations were that a couple of features will strongly correlate with employee attrition, while others less so. We expected an employee's income would significantly impact them staying at the company or not, with higher incomes seeing lower attrition and vice versa (inversely proportional). Related to this, we expected to see employee salary increase to be a strong predictor of attrition, with higher salary increases correlating with lower attrition rates. Some other correlations with attrition we expected to see were distance of commute (longer commute correlated with attrition), job and environment satisfaction (lower scores correlating with attrition), and work-life balance (lower scores correlated with higher attrition).

Our inspirations for this project were a number of employee attrition Tableau dashboards found on Tableau Public. See here, here, here, and here.

I. Exploratory Data Analysis (EDA) and Results

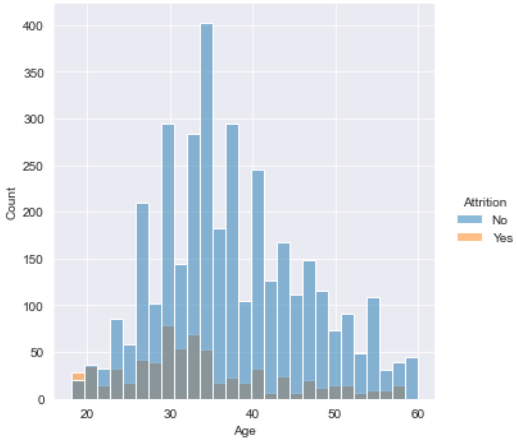
The raw data used in this project came from a case study on Kaggle called "HR Analytics Case Study". Three tables of the dataset were used, called `employee_survey_data`, `manager_survey_data`, and `general_data` based on the unique "EmployeeID" key. It was determined that these three datasets contained the most relevant data points from which to build the machine learning model.

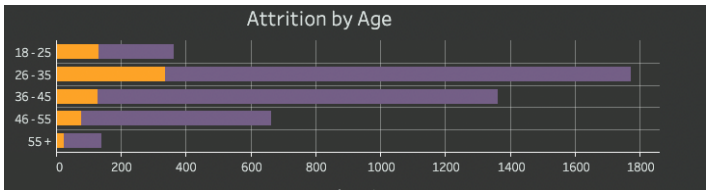
The CSVs were imported as data frames using Pandas and then the three data tables merged on the "EmployeeID" index. After that superfluous columns were dropped, particularly "EmployeeCount" and "Over18" columns. Next, the "MonthlyIncome" and "DistanceFromHome" columns were converted to USD and miles respectively. Following this a

.info() was called to see how many null values were present as well as the data types. After determining that 110 rows with null values would be dropped (out of 4410 rows) it was decided that this would be acceptable without compromising the integrity of the data. This cleaned dataset was saved as a separate CSV to keep the object data types for tableau visualizations.

The next step was to preprocess the data for a machine learning model by changing some of the data types from objects to integers. This was done by creating a list called “attrition cat” that held all data that were object data types. After that the OneHotEncoder function was used from the SKLearn library to give each unique object its own column where it would display 0 or 1 if it equaled the object. Once that was complete, this data was merged into the original data frame and saved to a separate CSV for machine learning.

Initial analysis of the cleaned data showed a couple of interesting, though perhaps not surprising, insights. First, younger employees (35 years and younger) left at higher rates than older employees, with those ages 18-25 years old leaving

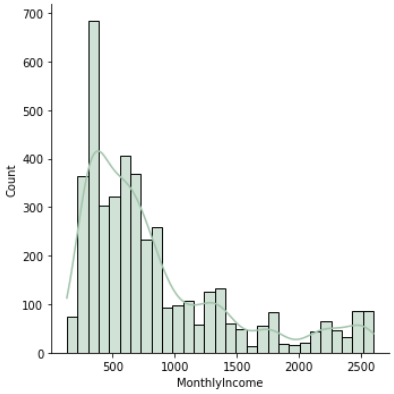




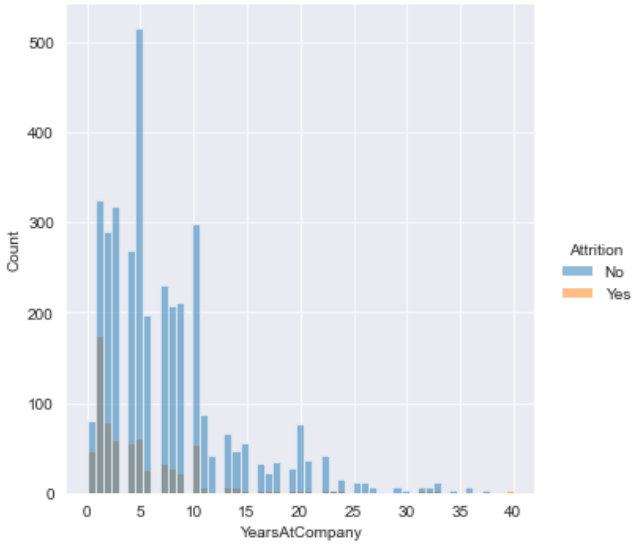
One would expect to see a relationship between attrition and income, and that is indeed what was seen during initial data exploration. The mean monthly income was **$200 higher** than the median, indicating right skew of the dataset (i.e. a few large incomes are pulling the average higher):

```
hr_df.MonthlyIncome.describe()
```

```
count    4300.000000
mean      845.777974
std       611.590186
min       131.170000
25%       380.380000
50%       641.680000
75%      1089.432500
max      2599.870000
Name: MonthlyIncome, dtype: float64
```



It was discovered that the majority of employees at the company have been there 10 years or less, with the highest attrition among employees at the company 1 year or less:



Based on the initial data exploration it was hypothesized that younger employees having been at the company 1 year or less with low monthly income would leave at the highest rates.

II. SQLite Database

After the EDA process and data transformation was complete, attention turned towards fitting the dataset into a relational database management system (RDBMS). The goal here was to have a database optimized for SQLite and minimal storage space.

The first task was to think about the schema and create an entity relationship diagram (ERD). The cleaned dataset was a single large table which required seeking out logical split points from which to build separate tables. It was determined to split the data into two primary tables and two smaller tables. The split point for the primary tables was to put information and demographics about employees in one table, and information about the workplace into another. The smaller tables were created by splitting the survey data into two separate tables.

Next, string values and columns that would be better represented numerically were sought out to save disk space. This resulted in six columns, three from each primary table, from which lookup tables were created.

The Final ERD can be seen here:

When working with SQLite and SQLAlchemy it became clear that the task of creating a database from scratch was outside the scope of this project, so extra time was taken to read the documentation and research the best way forward to carry out this task.

Using Jupyter Notebooks with Pandas and SQLAlchemy each table was created as a dataframe. The lookup tables were simplest and low hanging fruit, so this was a logical starting point.

Split HR_DF Into Tables Guided by ERD

Create Key Tables First

In [16]:
```
educationKey_df = pd.DataFrame()
educationKey_df['EducationID'] = [1,2,3,4,5]
educationKey_df['Education'] = ['Below College', 'College', 'Bachelor', 'Master', 'Doctor']
# educationKey_df['LastUpdated'] = pd.to_datetime('now')
educationKey_df['LastUpdated'] = datetime.now()
educationKey_df
```

Out[16]:

	EducationID	Education	LastUpdated
0	1	Below College	2022-08-11 22:36:35.741292
1	2	College	2022-08-11 22:36:35.741292
2	3	Bachelor	2022-08-11 22:36:35.741292
3	4	Master	2022-08-11 22:36:35.741292
4	5	Doctor	2022-08-11 22:36:35.741292

Next were the primary tables, which also required transformations of the columns that related to the lookup tables. After transcribing each value with its own line of code once, iterative scripts were constructed to transform the columns.

First, the `EmpWorkInfo` dataframe was created:

```
empWorkInfo_df = hr_df[['EmployeeID', 'Attrition','BusinessTravel','Department',
                        'JobLevel','JobRole','MonthlyIncome','PercentSalaryHike','StandardHours',
                        'StockOptionLevel','TrainingTimesLastYear','YearsAtCompany',
                        'YearsSinceLastPromotion','YearsWithCurrManager']]
empWorkInfo_df['EmpWorkTableID'] = range(1, len(empWorkInfo_df)+1)
empWorkInfo_df['LastUpdated'] = datetime.now()
empWorkInfo_df = empWorkInfo_df[['EmpWorkTableID','EmployeeID', 'Attrition','BusinessTravel','Department',
                        'JobLevel','JobRole','MonthlyIncome','PercentSalaryHike','StandardHours',
                        'StockOptionLevel','TrainingTimesLastYear','YearsAtCompany',
                        'YearsSinceLastPromotion','YearsWithCurrManager', 'LastUpdated']]
empWorkInfo_df.head(10)
```

Next, the lookup table was then used to transcribe the corresponding column.

```
for value in departmentKey_df.Department:
    print(value)
    empWorkInfo_df.Department.loc[empWorkInfo_df['Department'] == value]
        = departmentKey_df.loc[departmentKey_df['Department'] == value, 'DepartmentID'].to_list()[0]
    print(next)

empWorkInfo_df.head(10)
```

After double and triple checking to make sure each data frame matched the ERD (and finding many reasons to edit both), scripts were written to create the database. This was made slightly more simple because the ERD tool exported the schema into different formats. None of them were SQLAlchemy specific, but were able to be copied, pasted, and edited from there.

```
EmpWorkInfo = Table(
    'EmpWorkInfo', meta,
    Column("EmpWorkTableID", Integer, primary_key = True),
    Column("EmployeeID", Integer, ForeignKey('EmpPersonalInfo.EmployeeID')),
    Column("Attrition", String),
    Column("BusinessTravelID", String, ForeignKey('BusinessTravelKey.BusinessTravelID')),
    Column("DepartmentID", Integer, ForeignKey('DepartmentKey.DepartmentID')),
    Column("JobLevel", Integer),
    Column("JobRoleID", Integer, ForeignKey('JobRoleKey.JobRoleID')),
    Column("MonthlyIncome", Float),
    Column("PercentSalaryHike", Integer),
    Column("StandardHours", Integer),
    Column("StockOptionLevel", Integer),
    Column("TrainingTimesLastYear", Integer),
    Column("YearsAtCompany", Integer),
    Column("YearsSinceLastPromotion", Integer),
    Column("YearsWithCurrManager", Integer),
    Column("LastUpdated", DateTime),
    extend_existing=True,
)
```

One obstacle was the fact that SQLAlchemy requires users to individually import each data type that is to be used. Once the proper date time dependency was imported this was solved.

Once the SQLite database and its corresponding tables were correctly created, it was time to load in the data.

```
# populate tables with data from dataframes
empPersonalInfo_df.to_sql('EmpPersonalInfo', con=engine, if_exists='append', index=False)
empWorkInfo_df.to_sql('EmpWorkInfo', con=engine, if_exists='append', index=False)
educationKey_df.to_sql('EducationKey', con=engine, if_exists='append', index=False)
educationFieldKey_df.to_sql('EducationFieldKey', con=engine, if_exists='append', index=False)
marriageKey_df.to_sql('MarriageKey_df', con=engine, if_exists='append', index=False)
businessTravelKey_df.to_sql('BusinessTravelKey', con=engine, if_exists='append', index=False)
departmentKey_df.to_sql('DepartmentKey', con=engine, if_exists='append', index=False)
jobRoleKey_df.to_sql('JobRoleKey', con=engine, if_exists='append', index=False)
empSurvey_df.to_sql('EmpSurvey', con=engine, if_exists='append', index=False)
managerSurvey_df.to_sql('ManagerSurvey', con=engine, if_exists='append', index=False)
```

There was a lot of effort expended getting to this point, as each look at the ERD and the tables seemed to reveal another missed detail or something out of place to fix. Once the script was ready the data was loaded into the database, a new notebook was created, and successful queries were run.

In order to set up the Ploty page and the SQL exploration page multiple queries were written which resembled one another, though with different outputs for different uses. These were then moved into web application production for use on user-interactive pages.

```
query = f"""
SELECT
    Age,
    Attrition,
    MonthlyIncome
FROM
    EmpPersonalInfo
LEFT JOIN EmpWorkInfo ON
EmpPersonalInfo.EmployeeID = EmpWorkInfo.EmployeeID
WHERE Age >= {min_age}
    AND Age <= {max_age};
"""
```

A few other potentially useful queries:

```
department = '"Sales"'


query = f"""
        SELECT
            dk.Department,
            count(ew.EmployeeID)
        FROM
            EmpWorkInfo ew
        LEFT JOIN DepartmentKey dk ON
            ew.DepartmentID = dk.DepartmentID
        WHERE dk.Department = {department}
        """

df = pd.read_sql(query, conn_string)
df.head()
```

```
job_role = '"Research Scientist"'


query = f"""
        SELECT
            ep.EmployeeID,
            ep.Gender,
            jr.JobRole,
            ew.MonthlyIncome,
            ew.YearsAtCompany
        FROM
            EmpPersonalInfo ep
        LEFT JOIN EmpWorkInfo ew ON
            ep.EmployeeID = ew.EmployeeID
        LEFT JOIN JobRoleKey jr ON
            ew.JobRoleID = jr.JobRoleID
        WHERE jr.JobRole = {job_role}
        """

df = pd.read_sql(query, conn_string)
df.head()
```

	EmployeeID	Gender	JobRole	MonthlyIncome	YearsAtCompany
0	2	Female	Research Scientist	544.57	5
1	14	Male	Research Scientist	749.06	10
2	22	Male	Research Scientist	1256.71	10
3	23	Female	Research Scientist	279.24	5
4	26	Female	Research Scientist	883.87	10

III. Machine Learning Model Construction and Results

The process of deciding which machine learning model to use boiled down to which model produced the most accurate results. Several models were tested for accuracy and precision with the one with the least amount of mistakes picked for this project. Since the target value ("Attrition") was a binary variable, we used various logistic regression models to determine which would perform best for classification.

The first step for determining which model would be used for this project was to clean the data of all variables that did not have a significant impact on the overall composition of the dataset. Each of the variables had stat weights, where something like Age or Monthly Income had a much stronger importance for predictive power. The complete summarized stat weights, or "Data Importances" is listed as a resource on the github page for this project. What is listed below is the variables that were not cut from the model as they had a >1% impact on the overall results of the dataset: While no one category had a greater than 8% impact on the dataset, any of the stat weights that had a <1 % impact on the dataset were cut. Many of these variables had to do with the types of positions at the company or other factors that were deemed insignificant for the model's predictive power.

```
[(0.08089240121957782, 'Age'),
 (0.07229263793827248, 'TotalWorkingYears'),
 (0.06716741244236442, 'MonthlyIncome'),
 (0.06045724199487344, 'YearsAtCompany'),
 (0.05679799178458187, 'YearsWithCurrManager'),
 (0.04737367447256194, 'DistanceFromHome'),
 (0.047205867943362706, 'NumCompaniesWorked'),
 (0.04225577834063372, 'PercentSalaryHike'),
 (0.03779039656636137, 'TrainingTimesLastYear'),
 (0.03656613491328336, 'EmployeeID'),
 (0.03443000137134337, 'JobSatisfaction'),
 (0.033942080931259264, 'YearsSinceLastPromotion'),
 (0.032921658246131545, 'EnvironmentSatisfaction'),
 (0.029717469091213745, 'WorkLifeBalance'),
 (0.02598697683168786, 'JobLevel'),
 (0.025746894116803077, 'Education'),
```

Next was the question of which specific machine learning model should be used for this dataset that would make the least amount of mistakes. No matter which model is used, the data must be split into two groups: a “training” group of variables and a “test” group of variables. The training group consists of the original dataset being used to fit the parameters of the model. These variables are used to optimize the model to prepare it for a test group of variables, which will output the final model fit on the training dataset. This will result in a balanced accuracy score to rate how well the model did overall, factoring false positives and false negatives.

Oversampling was the first model that we used to test the dataset, where the minority class of variables were duplicated in order to match the majority class variables. This model ended up not being a good fit, since the dataset had a fairly even distribution of stat weights from 2-8%, leading to a balanced accuracy score of 65%. Undersampling had a similar issue, since the dataset was well distributed enough to not warrant the deletion of random majority class variables. This resulted in a 63% balanced accuracy score, slightly worse than the oversampling model. The combination model, which included elements of both undersampling and oversampling, fell square between the previous two models with a 64% balanced accuracy score.

The SMOTE model, or Synthetic Minority Oversampling TechniquE, is used to generate new minority class instances from the existing class to compensate for the disparity between the majority and minority classes. This again was not a very useful model, as it resulted in a balanced accuracy score of 59%, the worst model tested thus far.

However, the final model tested, the Random Forest Model, significantly outperformed the other models. This supervised machine learning model had a few advantages over the other models, one of which is increased average accuracy and efficiency compared to other models, because random forest models typically perform well with larger datasets. After training the model, the average of many different instances of the machine learning model is taken to determine the accuracy and precision of the model. In this particular instance, the random forest

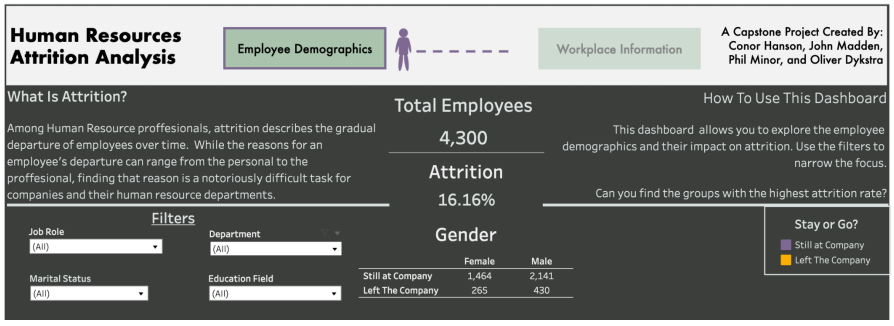
model had a balanced accuracy score of 96%, meaning 96% of the time it would be able to accurately predict if someone would leave their job or not based on the input variables.

IV. Data Visualization via Tableau

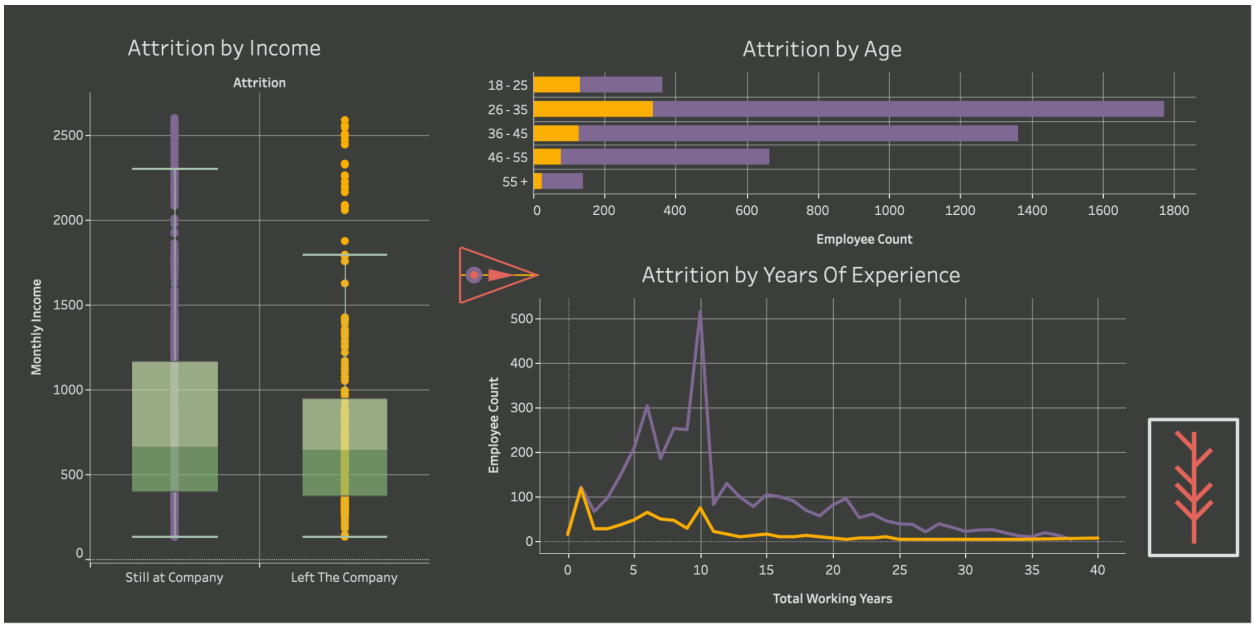
Using Tableau Public, two dashboards were created with three visualizations each. A filter forward approach was used in order to give the user more control over the data they wanted to see, while keeping a clean space that highlighted the visualizations. The dashboards were organized with the same logic as the database, and were split into an employee demographic view alongside a workplace view.

Using inspiration from Sibin Joseph and their public dashboard**,** buttons were created which provided an interactive and intuitive method to switch between the dashboards. This turned out to be one of the most challenging aspects of the dashboard, though in the end it was not as complicated as it first appeared. These buttons were able to be created that were imbued with a border and icon when active by simply editing the dashboards themselves instead of creating a calculated field or parameter, which was initially thought necessary.

Each dashboard has a nearly identical header which provides continuity between the dashboards, and includes context, dashboard selection buttons, filters, keys, and some basic top level numbers. This was a challenge to create and standardize. As edits were made to make the dashboards as close to identical as possible in order to provide a seamless transition between dashboards, Tableau's lack of functionality became apparent. It was discovered that numerous conversations on this issue were being had on Tableau forums, some nearly a decade old, criticizing this lack of functionality. This post suggests the ability to select multiple objects at a time so a user doesn't have to move each object individually is a good example of a feature that would make visualization creation a lot easier. In the future more care would be put into dashboard layout to create a template for other dashboards.



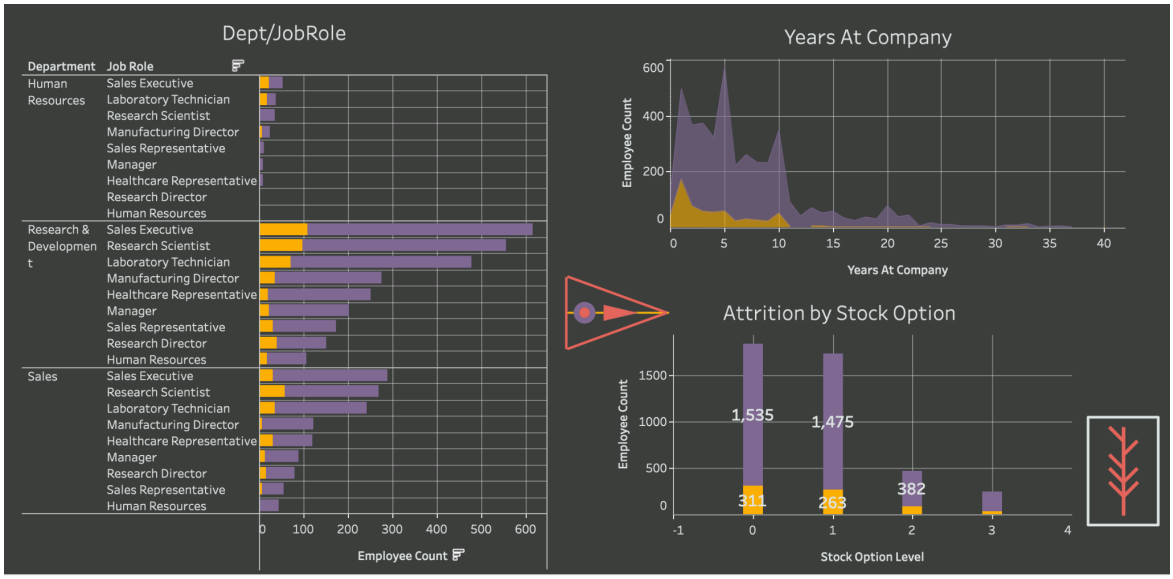
On the “Employee Demographics” dashboard attrition was visualized by “Income”, “Age”, and “Total Working Years” (renamed to “Years of Experience”). While these dimensions provided interesting insight into attrition as well as the opportunity to use different types of aesthetically compelling graphs, they are also the three most important features in the machine learning model. Here filters were provided for “Job Role, “Department”, “Marital Status”, and “Education Field”.



A box and whisker plot was used for “Attrition by Income” because income is the measure with the widest range and variability and, because of its statistical focus, a box and whisker plot captures and visualizes more valuable information from this measurement than other options.

For the “Attrition by Age” visualization buckets, or groups, were created in order to make the information more visually digestible.

On the “Workplace Information” dashboard attrition was visualized by “Department”, “Job Role”, “Years at Company”, and “Stock Option”. The filters provided on this dashboard are some selections from workplace surveys, “Job Involvement”, “Job Satisfaction”, “Work/Life Balance”, and “Environment Satisfaction”.



The Department and Job Role visualizations provide deep insight into the structure of the company, which allows a quick and simple determination of key trends and company make-up. Further, there is much value in the ability to quickly see the size of a department, the primary jobs within that department, and the attrition rate in each.

V. Web Application and Website

The website, hosted on Heroku[1], consists of 8 pages: a landing page, an attrition prediction page, a data visualization page (contains an interactive BI dashboard from Tableau), a Plotly visualization page which gives the user the ability to see attrition distribution by user-selected age ranges, a SQLite datatable page which can be filtered by the user, a documentation page containing this written report on the project, a page about the project team, and a page for works used, cited, or referenced during the course of this project. The color scheme for the website was chosen on coolors.co and contains a mix of muted colors that exude a slightly gloomy ethos, combined with some brighter and hopeful colors indicating that attrition may be avoidable.

[1] Heroku's free PAAS will end November 2022. We are currently seeking an alternative PAAS for hosting this web app.

There are 4 total web pages dedicated to various forms of data visualization and interaction. One of these pages is described above (Data Visualization via Tableau). A second page is an interactive graphing page, which uses Plotly to display a graph of attrition based on user-defined age groups. A third page is the database exploration page, which allows a user to select age groups, gender, and attrition to display (nearly) the entire database table from SQLite.

The fourth and most interesting page is dedicated to predicting employee attrition based on user inputs. A JavaScript script (logic.js) collects the user input values from the website and sends them off to a Flask app route as a JSON file via AJAX. The Flask app route ("/makepredictions") contains a function that parses the data and calls a helper function file (modelHelper.py) file to convert the JSON data via Pickle into an array which can then be run through the machine learning model loaded in via Pickle. The output of the machine learning model is then re-JSONified and, based on conditional arguments, the prediction displayed on the webpage via the JavaScript file (logic.js).

The Javascript logic.js file is responsible for gathering the user inputs for prediction and performing an AJAX request to the app.py file. This gets the output of the machine learning model and, based on a condition, will output the result to the webpage without it reloading.

The Python files are responsible for running the Flask app (app.py) and running the input prediction variables through the machine learning model (modelHelper.py)

The website was built using a free template from HTML5.com and repurposed according project needs. More pages were added beyond the base pages that came with the template. Colors were injected via the style.css file and applied throughout the website with SCSS. SCSS, or Sassy CSS language, is a CSS preprocessing language that allows a back-end web developer control over web elements beyond that of typical CSS. SCSS allows one to define style variables that can then be applied to those variables as they are found throughout the website. It does this by preprocessing the user-defined features in the SCSS file, translating the variables into CSS, and saving them in a typical CSS file to be utilized by the website.

For example, whereas CSS would define font and color with each element like the body, SCSS allows the user to define the font and color of a variable and then call that anywhere on a website. This greatly simplifies website development and allows greater customization to be done quickly.

SCSS:

```scss
$font-stack: Helvetica,
sans-serif;
$primary-color: #333;
body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

CSS:

```css
body {
  font: 100% Helvetica,
sans-serif;
  color: #333;
}
```

VI. Conclusions and Call to Action

A couple of preliminary conclusions may be drawn from this project, as well as a couple of recommendations for employers and employees. First, employees who are paid less than $1,000 per month and work at the company for less than one year are at the greatest risk of leaving. Perhaps employers can absorb this turnover, but it will almost certainly create a reputation of high-turnover and low pay, as well as impress upon employees the expectation that this situation is acceptable to the employer. Second, employees who commute a significant distance are less likely to stay at the company, especially if paid a lower monthly income. Helping give employees an incentive to stay at the company even if they have a long commute to work would be wise, both for the company's reputation and preventing operational disruptions. Third, younger employees are at a greater risk of leaving. However, given the limitations of the data (explained below) these insights need reinforcement from further data gathering, analysis, and prediction.

We recommend employers implement increased pay packages for employees, particularly new employees and younger employees, as this would likely prevent some (not all) attrition and

counteract potential business disruption. When employees feel supported and cared for they are more likely to stay at a company, which yields the increased potential for better company reputation and greater business stability.

VII. Limitations and Future Work on Employee Attrition

The limitations of the predictions garnered based on this data are substantial, albeit not such that they're totally worthless. The dataset is for a single year: 2015. While the dataset size of over 4,000 rows is good for prediction, because it only covers a single year it is inherently limited in its predictive power. For example, based on the prediction output, which is classified as attrition or not and given as a percentage, does this output mean the employee is likely to leave this year? Next year? Because the data is drawn from a single year it's not clear whether the employees leave immediately or the following year. Multiple years of attrition data would clarify trends and patterns over time to increase predictive granularity and power. Currently however, the predictive features allow an important insight into the reasons employees leave and what conversations management needs to be having if such attrition is to be slowed. Further, it would be even more useful for employers if they could see the average amount of time an employee stays with their company given these factors, which would require multiple years of data. This would allow for companies to be more proactive with their hiring and retention efforts if the results were timebound.

More longitudinal data collection is recommended for greater trend/pattern clarity and predictive power when seeking to understand employee attrition in the workplace. Data gathered from multiple consecutive calendar years would help fill out a clearer picture and strengthen the machine learning model, ultimately yielding a picture that would be most beneficial to employees and employers alike.