In [2]:
```
!pip install xgboost
```

```
Collecting xgboost
  Downloading xgboost-2.1.1-py3-none-macosx_10_15_x86_64.macosx_11_0_x86_64.macosx_12_0_
x86_64.whl (2.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.1/2.1 MB 3.3 MB/s eta 0:00:00a 0:00:01
Requirement already satisfied: numpy in /Users/induminajayathilaka/opt/anaconda3/lib/pyt
hon3.9/site-packages (from xgboost) (1.21.5)
Requirement already satisfied: scipy in /Users/induminajayathilaka/opt/anaconda3/lib/pyt
hon3.9/site-packages (from xgboost) (1.9.1)
Installing collected packages: xgboost
Successfully installed xgboost-2.1.1
```

In [2]:
```
#Importing packages
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

pd.set_option('display.max_columns', None)


from xgboost import XGBClassifier
from xgboost import XGBRegressor
from xgboost import plot_importance

from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier


from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score,\
f1_score, confusion_matrix, ConfusionMatrixDisplay, classification_report
from sklearn.metrics import roc_auc_score, roc_curve
from sklearn.tree import plot_tree


import pickle
```

In [3]:
```
# Loading the dataset

df0 = pd.read_csv("HR_comma_sep.csv")

# Displaying first few rows of the dataframe
df0.head()
```

Out[3]:

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_a
0	0.38	0.53	2	157	3	
1	0.80	0.86	5	262	6	
2	0.11	0.88	7	272	4	
3	0.72	0.87	5	223	5	
4	0.37	0.52	2	159	3	

Data Exploration (Initial EDA and data cleaning)

In [4]:
```python
df0.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14999 entries, 0 to 14998
Data columns (total 10 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   satisfaction_level     14999 non-null  float64
 1   last_evaluation        14999 non-null  float64
 2   number_project         14999 non-null  int64  
 3   average_montly_hours   14999 non-null  int64  
 4   time_spend_company     14999 non-null  int64  
 5   Work_accident          14999 non-null  int64  
 6   left                   14999 non-null  int64  
 7   promotion_last_5years  14999 non-null  int64  
 8   Department             14999 non-null  object 
 9   salary                 14999 non-null  object 
dtypes: float64(2), int64(6), object(2)
memory usage: 1.1+ MB
```

In [5]:
```python
df0.describe()
```

Out[5]:

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Wo
count	14999.000000	14999.000000	14999.000000	14999.000000	14999.000000	14
mean	0.612834	0.716102	3.803054	201.050337	3.498233	
std	0.248631	0.171169	1.232592	49.943099	1.460136	
min	0.090000	0.360000	2.000000	96.000000	2.000000	
25%	0.440000	0.560000	3.000000	156.000000	3.000000	
50%	0.640000	0.720000	4.000000	200.000000	3.000000	
75%	0.820000	0.870000	5.000000	245.000000	4.000000	
max	1.000000	1.000000	7.000000	310.000000	10.000000	

In [6]:
```python
df0.columns
```

Out[6]:
```
Index(['satisfaction_level', 'last_evaluation', 'number_project',
       'average_montly_hours', 'time_spend_company', 'Work_accident', 'left',
       'promotion_last_5years', 'Department', 'salary'],
      dtype='object')
```

In [7]:
```python
df0 = df0.rename(columns={'Work_accident': 'work_accident',
                          'average_montly_hours': 'average_monthly_hours',
                          'time_spend_company': 'tenure',
                          'Department': 'department'})

# Displaying all column names after the update
df0.columns
```

Out[7]:
```
Index(['satisfaction_level', 'last_evaluation', 'number_project',
       'average_monthly_hours', 'tenure', 'work_accident', 'left',
       'promotion_last_5years', 'department', 'salary'],
      dtype='object')
```

In [8]:
```python
#Checking for missing values
df0.isna().sum()
```

Out[8]:
```
satisfaction_level       0
last_evaluation          0
number_project           0
average_monthly_hours    0
tenure                   0
work_accident            0
left                     0
promotion_last_5years    0
department               0
salary                   0
dtype: int64
```

In [9]:
```
#Checking for duplicated values
df0.duplicated().sum()
```

Out[9]:
```
3008
```

In [10]:
```
df0[df0.duplicated()].head()
```

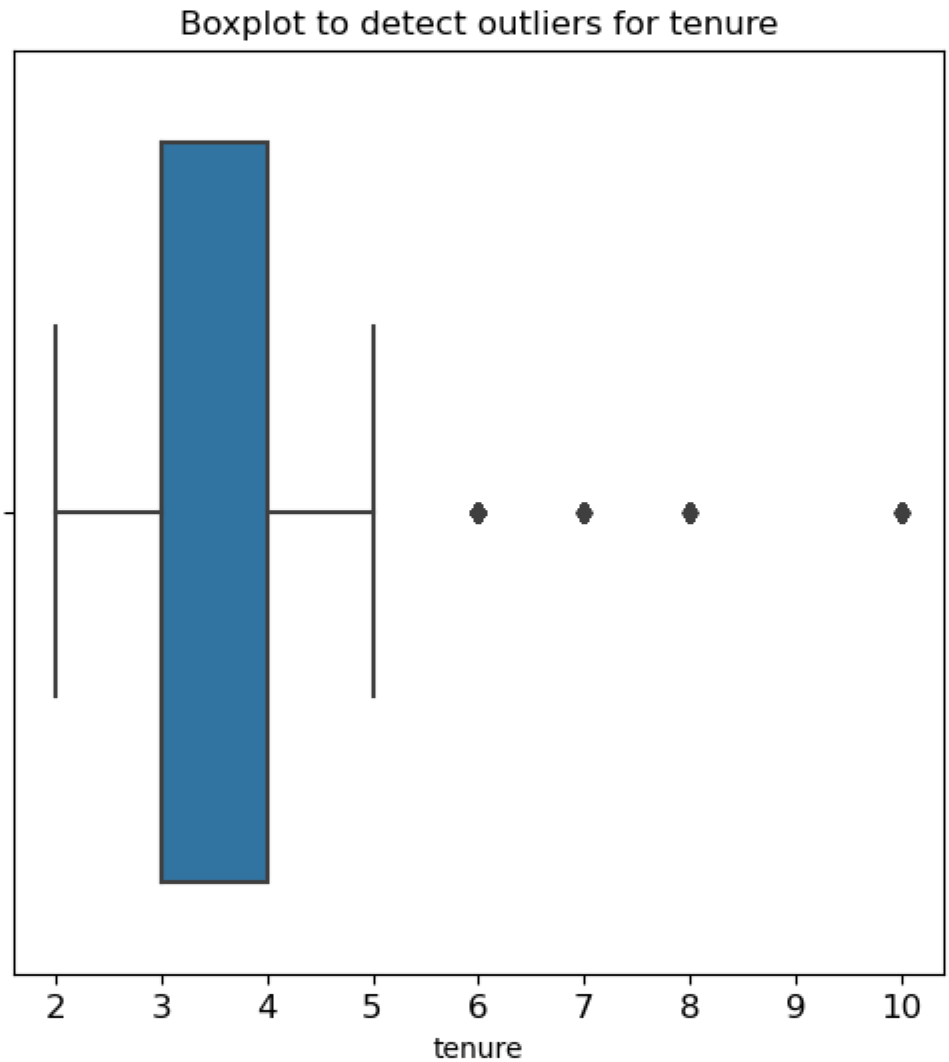
Out[10]:

	satisfaction_level	last_evaluation	number_project	average_monthly_hours	tenure	work_accident	l
396	0.46	0.57	2	139	3	0	
866	0.41	0.46	2	128	3	0	
1317	0.37	0.51	2	127	3	0	
1368	0.41	0.52	2	132	3	0	
1461	0.42	0.53	2	142	3	0	

In [11]:
```
#With several continuous variables across 10 columns, it seems very unlikely that these
```

In [12]:
```
# Droping duplicates and saving resulting dataframe in a new variable as needed

df1 = df0.drop_duplicates(keep='first')
```

In [13]:
```
# Creating a boxplot to visualize distribution of `tenure` and detect any outliers
plt.figure(figsize=(6,6))
plt.title('Boxplot to detect outliers for tenure', fontsize=12)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
sns.boxplot(x=df1['tenure'])
plt.show()
```



In [14]:

```
# Computing the 25th percentile value in `tenure`
percentile25 = df1['tenure'].quantile(0.25)

# Computing the 75th percentile value in `tenure`
percentile75 = df1['tenure'].quantile(0.75)

# Computing the interquartile range in `tenure`
iqr = percentile75 - percentile25

# Defining the upper limit and lower limit for non-outlier values in `tenure`
upper_limit = percentile75 + 1.5 * iqr
lower_limit = percentile25 - 1.5 * iqr
print("Lower limit:", lower_limit)
print("Upper limit:", upper_limit)

# Identifying subset of data containing outliers in `tenure`
outliers = df1[(df1['tenure'] > upper_limit) | (df1['tenure'] < lower_limit)]

# Counting how many rows in the data contain outliers in `tenure`
print("Number of rows in the data containing outliers in `tenure`:", len(outliers))
```

```
Lower limit: 1.5
Upper limit: 5.5
Number of rows in the data containing outliers in `tenure`: 824
```

In [15]:

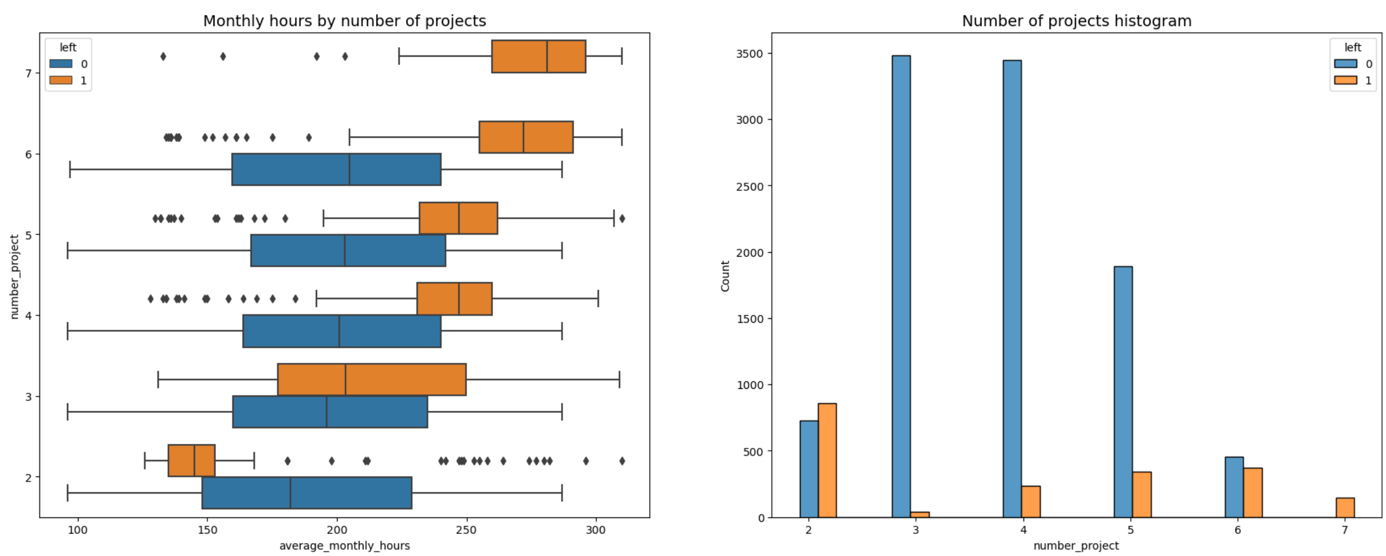
```
# Getting numbers of people who left vs. stayed

print(df1['left'].value_counts())
print()
```

```
# Getting percentages of people who left vs. stayed

print(df1['left'].value_counts(normalize=True))
```

```
0    10000
1     1991
Name: left, dtype: int64

0    0.833959
1    0.166041
Name: left, dtype: float64
```

In [16]:

```
# Setting figure and axes
fig, ax = plt.subplots(1, 2, figsize = (22,8))

# Creating boxplot showing `average_monthly_hours` distributions for `number_project`, c
sns.boxplot(data=df1, x='average_monthly_hours', y='number_project', hue='left', orient=
ax[0].invert_yaxis()
ax[0].set_title('Monthly hours by number of projects', fontsize='14')

# Creating histogram showing distribution of `number_project`, comparing employees who s
tenure_stay = df1[df1['left']==0]['number_project']
tenure_left = df1[df1['left']==1]['number_project']
sns.histplot(data=df1, x='number_project', hue='left', multiple='dodge', shrink=2, ax=ax
ax[1].set_title('Number of projects histogram', fontsize='14')

# Displaying the plots
plt.show()
```



It might be natural that people who work on more projects would also work longer hours. This appears to be the case here, with the mean hours of each group (stayed and left) increasing with number of projects worked. However, a few things stand out from this plot.

There are two groups of employees who left the company: (A) those who worked considerably less than their peers with the same number of projects (B) those who worked much more. Of those in group A, it's possible that they were fired. It's also possible that this group includes employees who had already given their notice and were assigned fewer hours because they were already on their way out the door. For those in group B, it's reasonable to infer that they probably quit. The folks in group B likely contributed a lot to the projects they worked in; they might have been the largest contributors to their projects.

Everyone with seven projects left the company, and the interquartile ranges of this group and those who left with six projects was ~255–295 hours/month—much more than any other group.

The optimal number of projects for employees to work on seems to be 3–4. The ratio of left/stayed is very small for these cohorts.

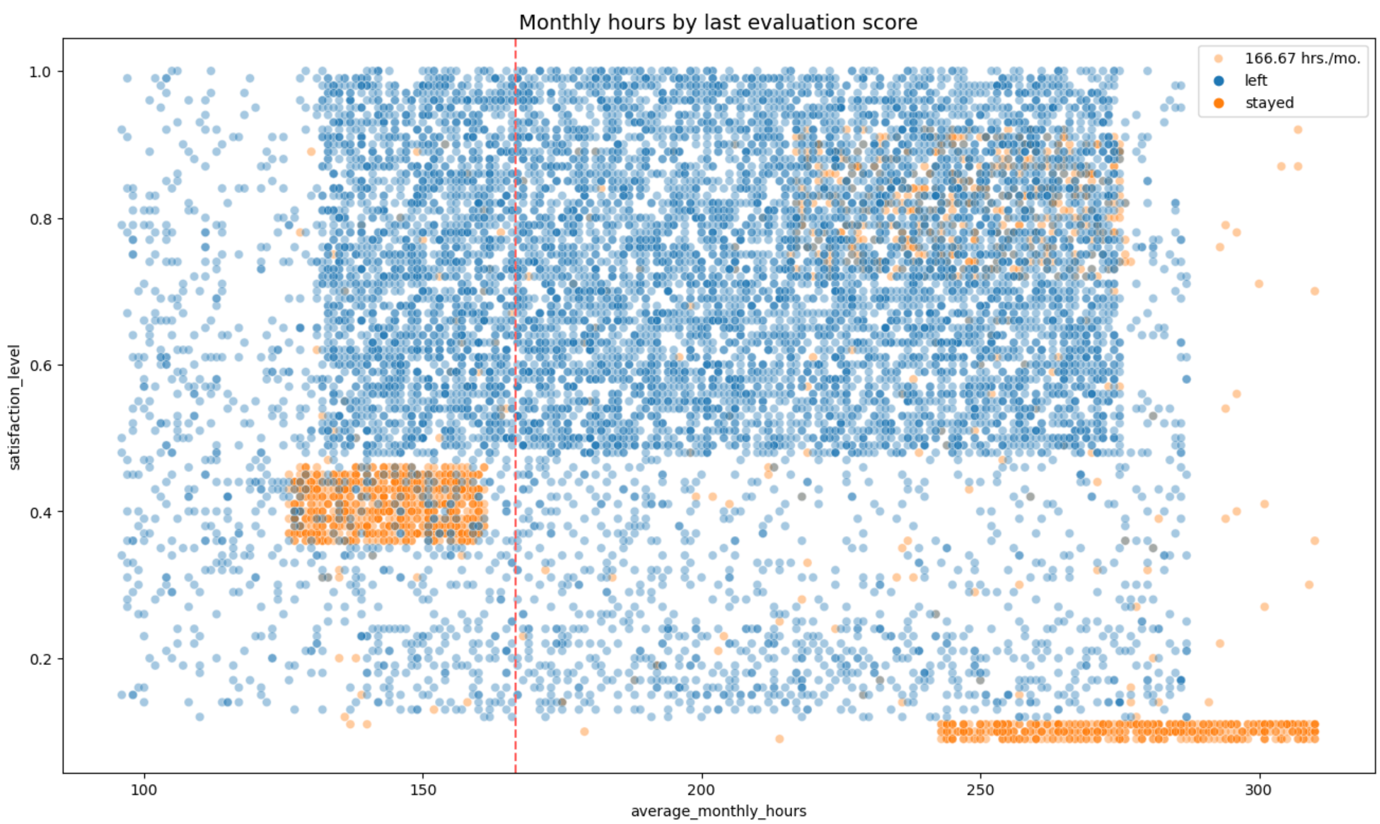
If we assume a work week of 40 hours and two weeks of vacation per year, then the average number of working hours per month of employees working Monday–Friday = 50 weeks * 40 hours per week / 12 months = 166.67 hours per month. This means that, aside from the employees who worked on two projects, every group—even those who didn't leave the company—worked considerably more hours than this. It seems that employees here are overworked.

In [17]:
```
# Getting value counts of stayed/left for employees with 7 projects
df1[df1['number_project']==7]['left'].value_counts()
```

Out[17]:
```
1    145
Name: left, dtype: int64
```

This confirms that all employees with 7 projects did leave.

In [18]:
```
# Creating scatterplot of `average_monthly_hours` versus `satisfaction_level`, comparing
plt.figure(figsize=(16, 9))
sns.scatterplot(data=df1, x='average_monthly_hours', y='satisfaction_level', hue='left',
plt.axvline(x=166.67, color='#ff6361', label='166.67 hrs./mo.', ls='--')
plt.legend(labels=['166.67 hrs./mo.', 'left', 'stayed'])
plt.title('Monthly hours by last evaluation score', fontsize='14');
```



The scatterplot above shows that there was a sizeable group of employees who worked ~240–315 hours per month. 315 hours per month is over 75 hours per week for a whole year. It's likely this is related to their satisfaction levels being close to zero.

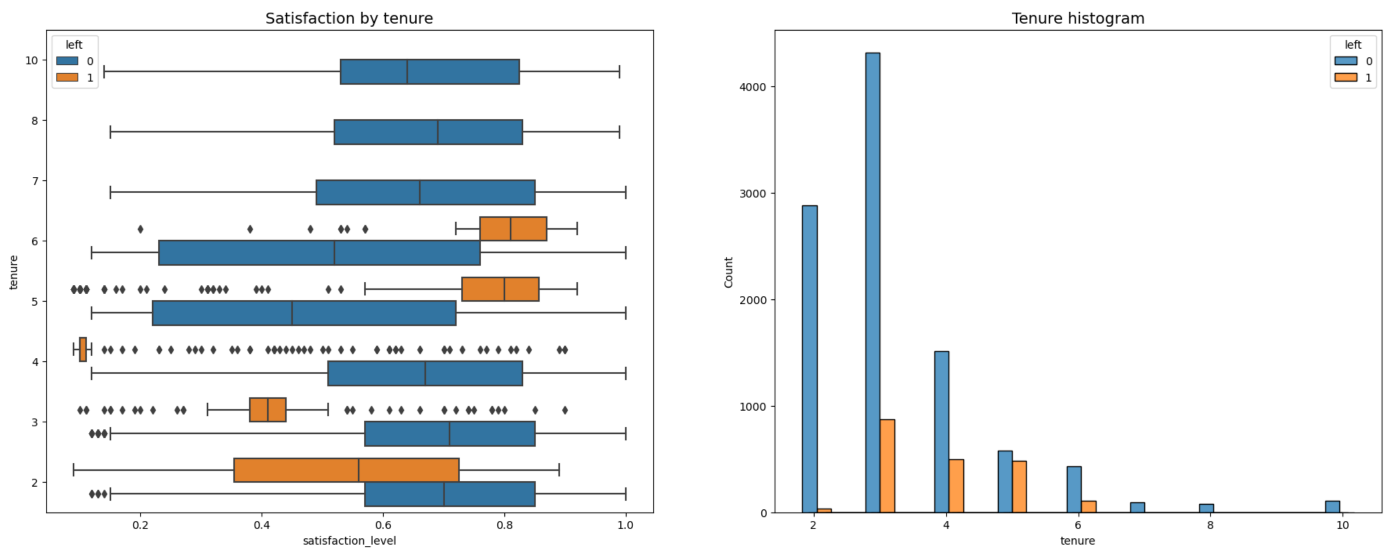
The plot also shows another group of people who left, those who had more normal working hours. Even so, their satisfaction was only around 0.4. It's difficult to speculate about why they might have left. It's possible they felt pressured to work more, considering so many of their peers worked more. And that pressure could have lowered their satisfaction levels.

Finally, there is a group who worked ~210–280 hours per month, and they had satisfaction levels ranging ~0.7–0.9.

Note the strange shape of the distributions here. This is indicative of data manipulation or synthetic data.

For the next visualization, it might be interesting to visualize satisfaction levels by tenure.

In [19]:
```
# Set figure and axes
fig, ax = plt.subplots(1, 2, figsize = (22,8))

# Creating boxplot showing distributions of `satisfaction_level` by tenure, comparing em
sns.boxplot(data=df1, x='satisfaction_level', y='tenure', hue='left', orient="h", ax=ax[
ax[0].invert_yaxis()
ax[0].set_title('Satisfaction by tenure', fontsize='14')

# Creating histogram showing distribution of `tenure`, comparing employees who stayed ve
tenure_stay = df1[df1['left']==0]['tenure']
tenure_left = df1[df1['left']==1]['tenure']
sns.histplot(data=df1, x='tenure', hue='left', multiple='dodge', shrink=5, ax=ax[1])
ax[1].set_title('Tenure histogram', fontsize='14')

plt.show();
```



There are many observations you could make from this plot.

Employees who left fall into two general categories: dissatisfied employees with shorter tenures and very satisfied employees with medium-length tenures. Four-year employees who left seem to have an unusually low satisfaction level. It's worth investigating changes to company policy that might have affected people specifically at the four-year mark, if possible. The longest-tenured employees didn't leave. Their satisfaction levels aligned with those of newer employees who stayed. The histogram shows that there are relatively few longer-tenured employees. It's possible that they're the higher-ranking, higher-paid employees.

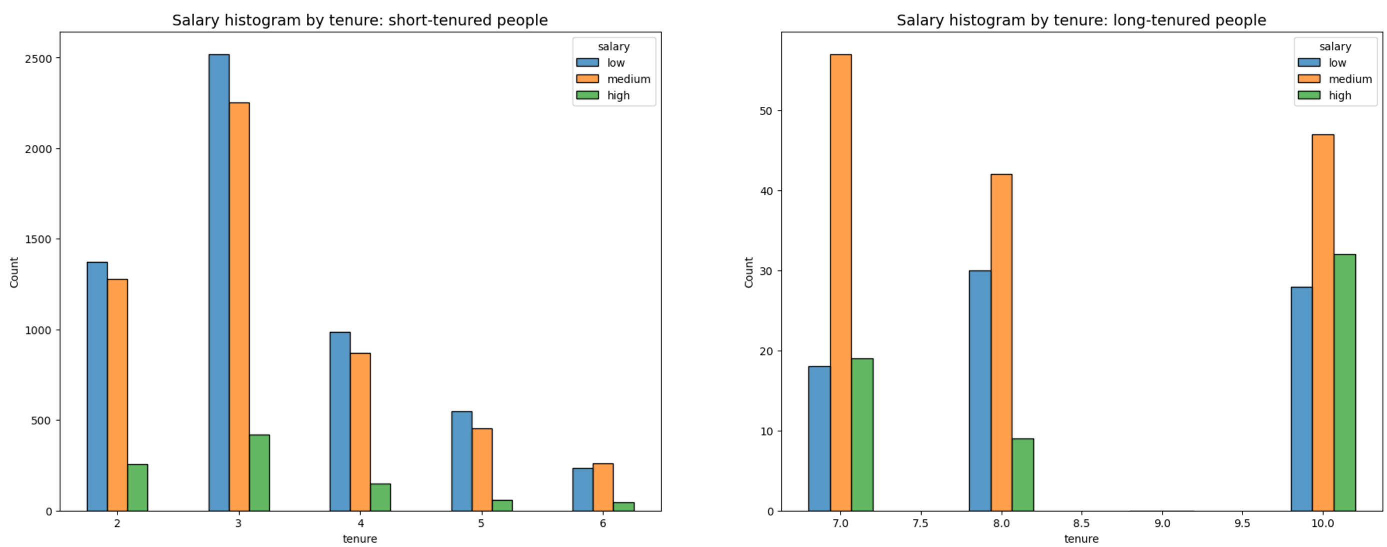
In [20]:
```
# Calculating mean and median satisfaction scores of employees who left and those who st
df1.groupby(['left'])['satisfaction_level'].agg([np.mean,np.median])
```

Out[20]:

left	mean	median
0	0.667365	0.69
1	0.440271	0.41

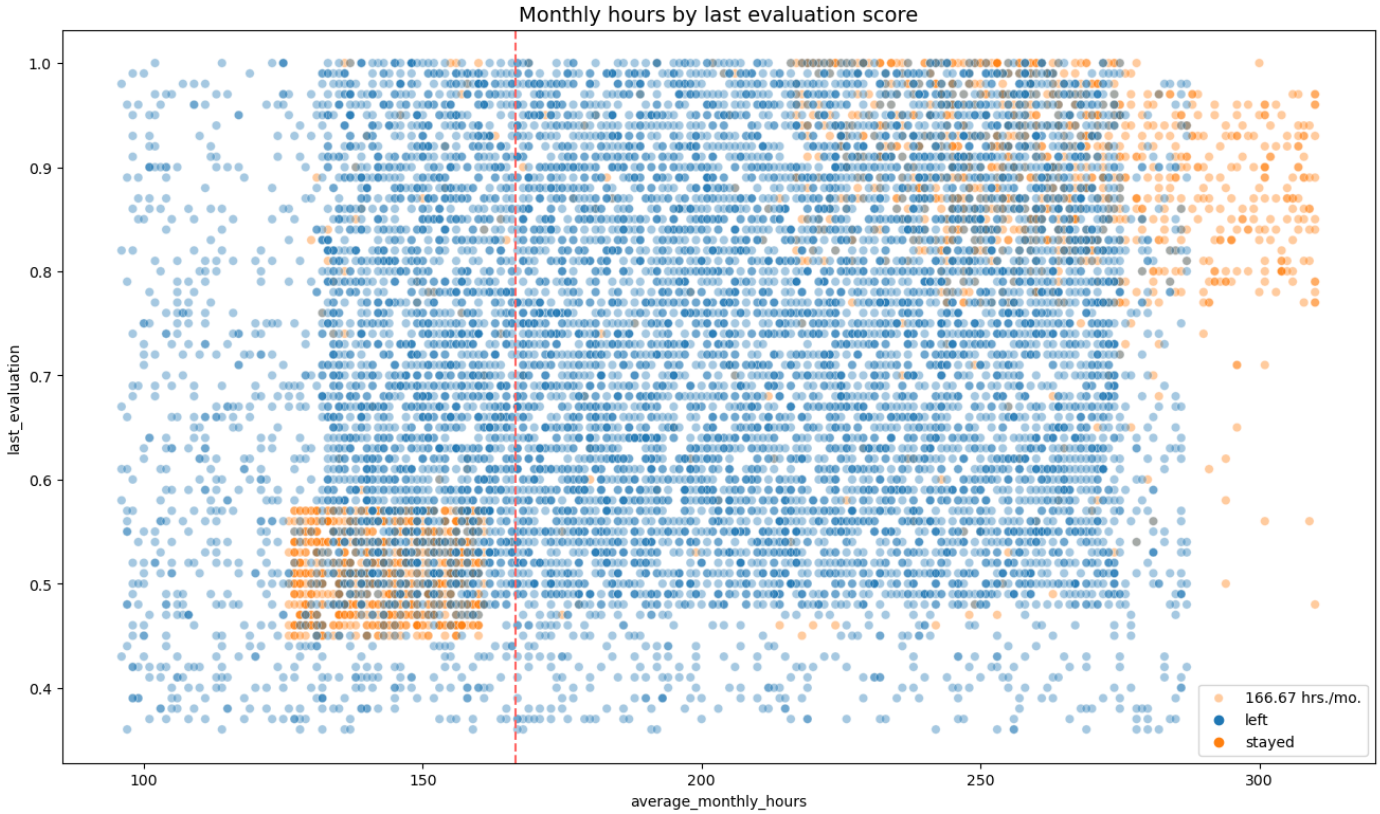
As expected, the mean and median satisfaction scores of employees who left are lower than those of employees who stayed. Interestingly, among employees who stayed, the mean satisfaction score appears to be slightly below the median score. This indicates that satisfaction levels among those who stayed might be skewed to the left.

In [21]:
```
# Setting figure and axes
fig, ax = plt.subplots(1, 2, figsize = (22,8))

# Defining short-tenured employees
tenure_short = df1[df1['tenure'] < 7]

# Defining long-tenured employees
tenure_long = df1[df1['tenure'] > 6]

# Plotting short-tenured histogram
sns.histplot(data=tenure_short, x='tenure', hue='salary', discrete=1,
             hue_order=['low', 'medium', 'high'], multiple='dodge', shrink=.5, ax=ax[0])
ax[0].set_title('Salary histogram by tenure: short-tenured people', fontsize='14')

# Plotting long-tenured histogram
sns.histplot(data=tenure_long, x='tenure', hue='salary', discrete=1,
             hue_order=['low', 'medium', 'high'], multiple='dodge', shrink=.4, ax=ax[1])
ax[1].set_title('Salary histogram by tenure: long-tenured people', fontsize='14');
```



The plots above show that long-tenured employees were not disproportionately comprised of higher-paid employees.

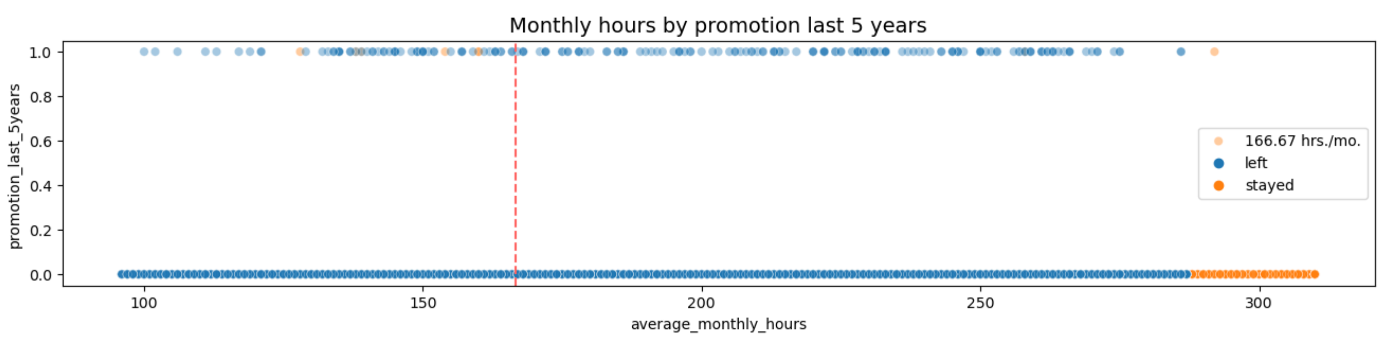
In [22]:
```
# Creating scatterplot of `average_monthly_hours` versus `last_evaluation`
plt.figure(figsize=(16, 9))
sns.scatterplot(data=df1, x='average_monthly_hours', y='last_evaluation', hue='left', al
plt.axvline(x=166.67, color='#ff6361', label='166.67 hrs./mo.', ls='--')
plt.legend(labels=['166.67 hrs./mo.', 'left', 'stayed'])
plt.title('Monthly hours by last evaluation score', fontsize='14');
```



The following observations can be made from the scatterplot above:

The scatterplot indicates two groups of employees who left: overworked employees who performed very well and employees who worked slightly under the nominal monthly average of 166.67 hours with lower evaluation scores. There seems to be a correlation between hours worked and evaluation score. There isn't a high percentage of employees in the upper left quadrant of this plot; but working long hours doesn't guarantee a good evaluation score. Most of the employees in this company work well over 167 hours per month.

In [23]:
```
# Creating plot to examine relationship between `average_monthly_hours` and `promotion_l
plt.figure(figsize=(16, 3))
sns.scatterplot(data=df1, x='average_monthly_hours', y='promotion_last_5years', hue='lef
plt.axvline(x=166.67, color='#ff6361', ls='--')
plt.legend(labels=['166.67 hrs./mo.', 'left', 'stayed'])
plt.title('Monthly hours by promotion last 5 years', fontsize='14');
```



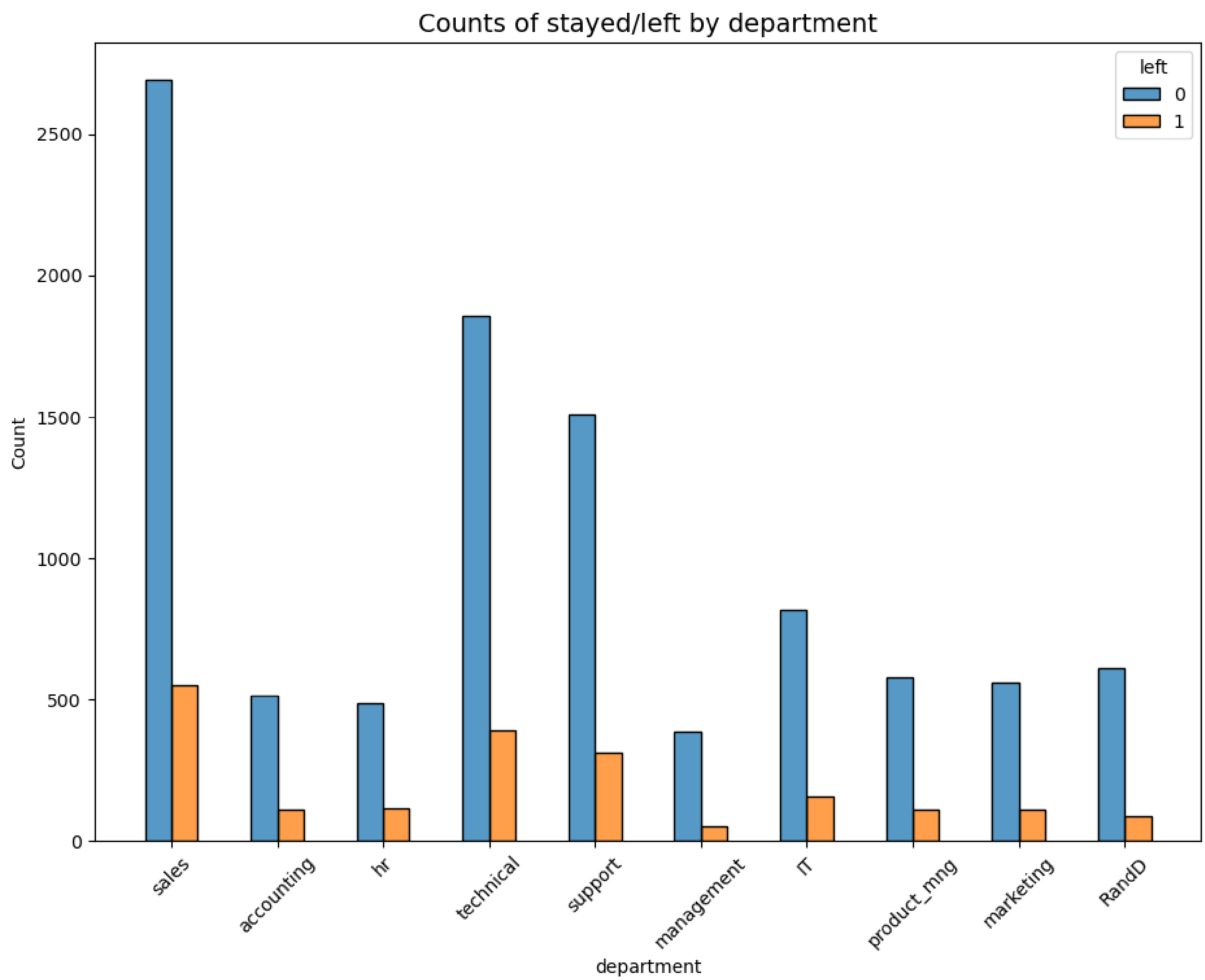
The plot above shows the following:

very few employees who were promoted in the last five years left very few employees who worked the most hours were promoted all of the employees who left were working the longest hours

In [24]:
```
# Displaying counts for each department
df1["department"].value_counts()
```

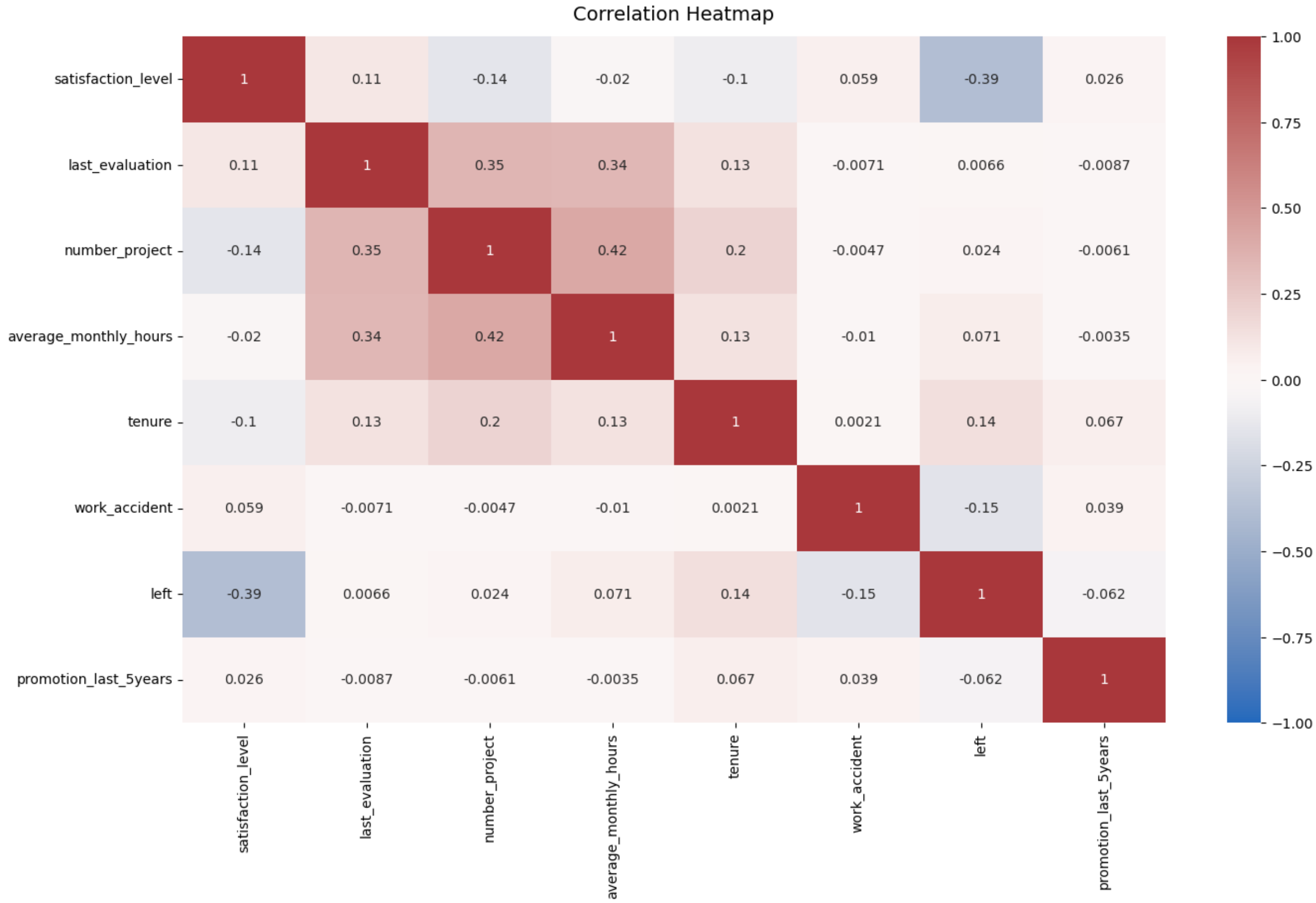
Out[24]:
```
sales           3239
technical       2244
support         1821
IT               976
RandD            694
product_mng      686
marketing        673
accounting       621
hr               601
management       436
Name: department, dtype: int64
```

In [25]:
```
# Creating stacked histogram to compare department distribution of employees who left to
plt.figure(figsize=(11,8))
sns.histplot(data=df1, x='department', hue='left', discrete=1,
             hue_order=[0, 1], multiple='dodge', shrink=.5)
plt.xticks(rotation='45')
plt.title('Counts of stayed/left by department', fontsize=14);
```



There doesn't seem to be any department that differs significantly in its proportion of employees who left to those who stayed.

In [26]:
```
# Plotting a correlation heatmap
plt.figure(figsize=(16, 9))
heatmap = sns.heatmap(df0.corr(), vmin=-1, vmax=1, annot=True, cmap=sns.color_palette("v
heatmap.set_title('Correlation Heatmap', fontdict={'fontsize':14}, pad=12);
```



The correlation heatmap confirms that the number of projects, monthly hours, and evaluation scores all have some positive correlation with each other, and whether an employee leaves is negatively correlated with their satisfaction level.

Insights from Data Exploration

Insights It appears that employees are leaving the company as a result of poor management. Leaving is tied to longer working hours, many projects, and generally lower satisfaction levels. It can be ungratifying to work long hours and not receive promotions or good evaluation scores. There's a sizeable group of employees at this company who are probably burned out. It also appears that if an employee has spent more than six years at the company, they tend not to leave.

Model Building

In [27]:

```
# Copying the dataframe
df_enc = df1.copy()

# Encoding the `salary` column as an ordinal numeric category
df_enc['salary'] = (
    df_enc['salary'].astype('category')
    .cat.set_categories(['low', 'medium', 'high'])
    .cat.codes
)

# Dummying encode the `department` column
df_enc = pd.get_dummies(df_enc, drop_first=False)
```

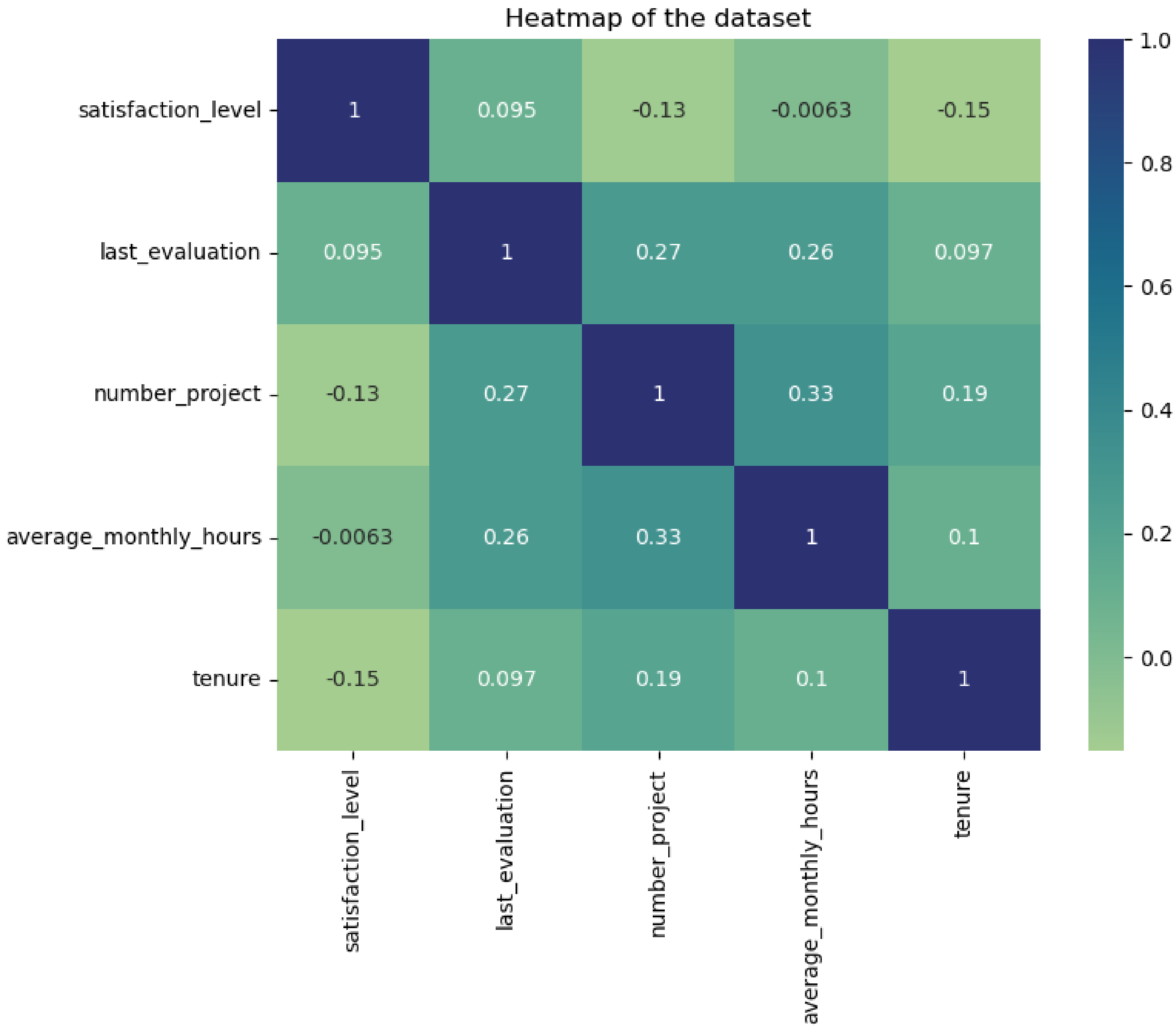
```python
# Displaying the new dataframe
df_enc.head()
```

Out[27]:

	satisfaction_level	last_evaluation	number_project	average_monthly_hours	tenure	work_accident	left
0	0.38	0.53	2	157	3	0	1
1	0.80	0.86	5	262	6	0	1
2	0.11	0.88	7	272	4	0	1
3	0.72	0.87	5	223	5	0	1
4	0.37	0.52	2	159	3	0	1

In [28]:

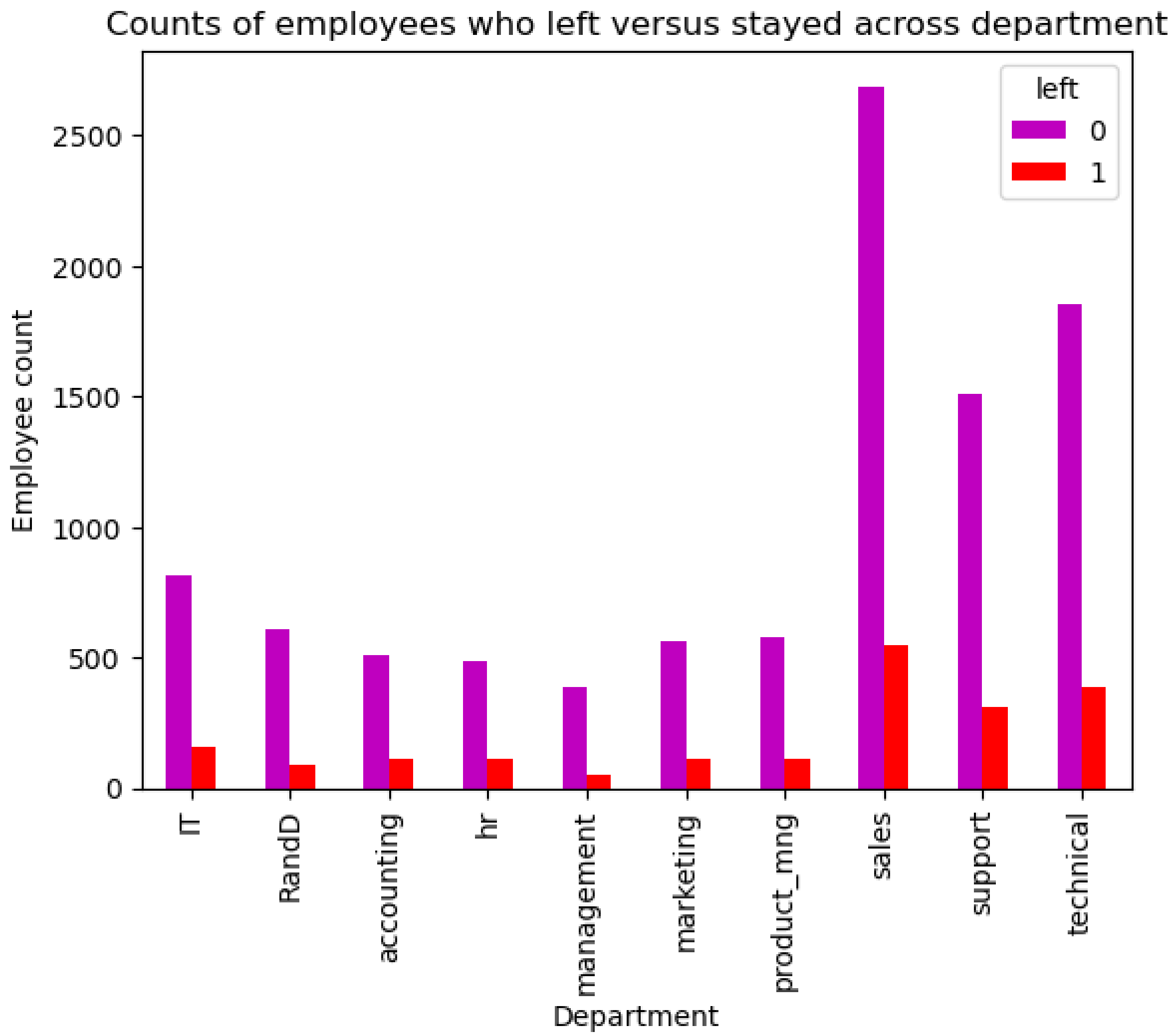
```python
# Creating a heatmap to visualize how correlated variables are
plt.figure(figsize=(8, 6))
sns.heatmap(df_enc[['satisfaction_level', 'last_evaluation', 'number_project', 'average_
            .corr(), annot=True, cmap="crest")
plt.title('Heatmap of the dataset')
plt.show()
```



In [29]:

```python
# Creating a stacked bart plot to visualize number of employees across department, compa
# In the legend, 0 (purple color) represents employees who did not leave, 1 (red color)
pd.crosstab(df1['department'], df1['left']).plot(kind ='bar',color='mr')
plt.title('Counts of employees who left versus stayed across department')
```

```
plt.ylabel('Employee count')
plt.xlabel('Department')
plt.show()
```



In [30]:

```
# Selecting rows without outliers in `tenure` and save resulting dataframe in a new vari
df_logreg = df_enc[(df_enc['tenure'] >= lower_limit) & (df_enc['tenure'] <= upper_limit)

# Displaying first few rows of new dataframe
df_logreg.head()
```

Out[30]:

	satisfaction_level	last_evaluation	number_project	average_monthly_hours	tenure	work_accident	left
0	0.38	0.53	2	157	3	0	1
2	0.11	0.88	7	272	4	0	1
3	0.72	0.87	5	223	5	0	1
4	0.37	0.52	2	159	3	0	1
5	0.41	0.50	2	153	3	0	1

In [31]:

```
# Isolating the outcome variable
y = df_logreg['left']

# Displaying first few rows of the outcome variable
y.head()
```

Out[31]:
```
0    1
2    1
3    1
4    1
5    1
Name: left, dtype: int64
```

In [32]:
```
# Selecting the features
X = df_logreg.drop('left', axis=1)

# Displaying the first few rows of the selected features
X.head()
```

Out[32]:

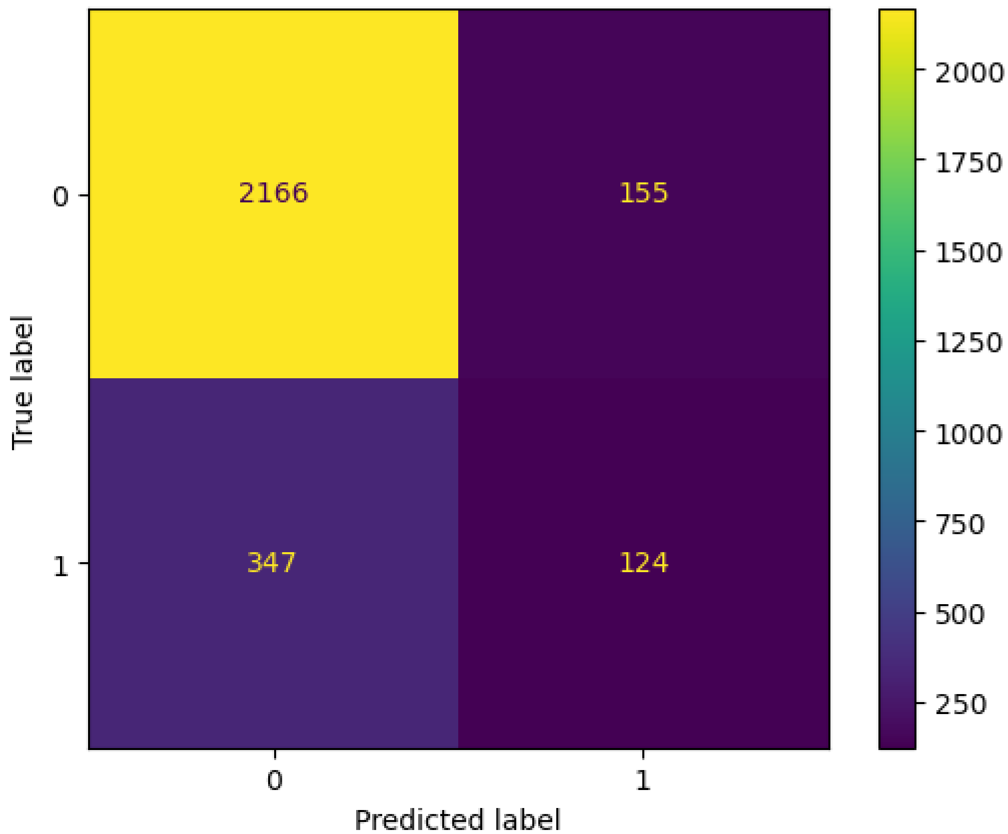
	satisfaction_level	last_evaluation	number_project	average_monthly_hours	tenure	work_accident	pron
0	0.38	0.53	2	157	3	0	
2	0.11	0.88	7	272	4	0	
3	0.72	0.87	5	223	5	0	
4	0.37	0.52	2	159	3	0	
5	0.41	0.50	2	153	3	0	

In [33]:
```
# Splitting the data into training set and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, stratify=y, ra
```

In [34]:
```
# Constructing a logistic regression model and fit it to the training dataset
log_clf = LogisticRegression(random_state=42, max_iter=500).fit(X_train, y_train)
```

In [35]:
```
# Using the logistic regression model to get predictions on the test set
y_pred = log_clf.predict(X_test)
```

In [36]:
```
# Computing values for confusion matrix
log_cm = confusion_matrix(y_test, y_pred, labels=log_clf.classes_)

# Creating display of confusion matrix
log_disp = ConfusionMatrixDisplay(confusion_matrix=log_cm,
                                  display_labels=log_clf.classes_)

# Plotting confusion matrix
log_disp.plot(values_format='')

# Displaying plot
plt.show()
```



In [37]:
```
df_logreg['left'].value_counts(normalize=True)
```

Out[37]:
```
0    0.831468
1    0.168532
Name: left, dtype: float64
```

There is an approximately 83%-17% split. So the data is not perfectly balanced, but it is not too imbalanced. If it was more severely imbalanced, we might want to resample the data to make it more balanced. In this case, we can use this data without modifying the class balance and continue evaluating the model.

In [38]:
```
# Creating classification report for logistic regression model
target_names = ['Predicted would not leave', 'Predicted would leave']
print(classification_report(y_test, y_pred, target_names=target_names))
```

```
                           precision    recall  f1-score   support

Predicted would not leave       0.86      0.93      0.90      2321
    Predicted would leave       0.44      0.26      0.33       471

                 accuracy                           0.82      2792
                macro avg       0.65      0.60      0.61      2792
             weighted avg       0.79      0.82      0.80      2792
```

The classification report above shows that the logistic regression model achieved a precision of 79%, recall of 82%, f1-score of 80% (all weighted averages), and accuracy of 82%. However, if it's most important to predict employees who leave, then the scores are significantly lower.

Tree-based model

In [39]:
```
y = df_enc['left']
X = df_enc.drop('left', axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, stratify=y, ra
```

In [40]:
```
tree = DecisionTreeClassifier(random_state=0)

# Assignning a dictionary of hyperparameters to search over
cv_params = {'max_depth':[4, 6, 8, None],
             'min_samples_leaf': [2, 5, 1],
             'min_samples_split': [2, 4, 6]
             }

# Assignning a dictionary of scoring metrics to capture
scoring = {'accuracy', 'precision', 'recall', 'f1', 'roc_auc'}


tree1 = GridSearchCV(tree, cv_params, scoring=scoring, cv=4, refit='roc_auc')
```

In [41]:
```
tree1.fit(X_train, y_train)
```

Out[41]:
```
GridSearchCV(cv=4, estimator=DecisionTreeClassifier(random_state=0),
             param_grid={'max_depth': [4, 6, 8, None],
                         'min_samples_leaf': [2, 5, 1],
                         'min_samples_split': [2, 4, 6]},
             refit='roc_auc',
             scoring={'f1', 'recall', 'precision', 'accuracy', 'roc_auc'})
```

In [42]:
```
# Checking best parameters
tree1.best_params_
```

Out[42]:
```
{'max_depth': 4, 'min_samples_leaf': 5, 'min_samples_split': 2}
```

In [43]:
```
# Checking best AUC score on CV
tree1.best_score_
```

Out[43]:
```
0.969819392792457
```

This is a strong AUC score, which shows that this model can predict employees who will leave very well.

In [44]:
```
#Extracting scores
def make_results(model_name:str, model_object, metric:str):
    '''
    Arguments:
        model_name (string): what you want the model to be called in the output table
        model_object: a fit GridSearchCV object
        metric (string): precision, recall, f1, accuracy, or auc

    Returns a pandas df with the F1, recall, precision, accuracy, and auc scores
    for the model with the best mean 'metric' score across all validation folds.
    '''

    # Create dictionary that maps input metric to actual metric name in GridSearchCV
    metric_dict = {'auc': 'mean_test_roc_auc',
                   'precision': 'mean_test_precision',
                   'recall': 'mean_test_recall',
                   'f1': 'mean_test_f1',
                   'accuracy': 'mean_test_accuracy'
                  }

    # Get all the results from the CV and put them in a df
    cv_results = pd.DataFrame(model_object.cv_results_)
```

```
    # Isolate the row of the df with the max(metric) score
    best_estimator_results = cv_results.iloc[cv_results[metric_dict[metric]].idxmax(), :

    # Extract Accuracy, precision, recall, and f1 score from that row
    auc = best_estimator_results.mean_test_roc_auc
    f1 = best_estimator_results.mean_test_f1
    recall = best_estimator_results.mean_test_recall
    precision = best_estimator_results.mean_test_precision
    accuracy = best_estimator_results.mean_test_accuracy

    # Create table of results
    table = pd.DataFrame()
    table = pd.DataFrame({'model': [model_name],
                          'precision': [precision],
                          'recall': [recall],
                          'F1': [f1],
                          'accuracy': [accuracy],
                          'auc': [auc]
                        })

    return table
```

In [45]:

```
# Getting all CV scores
tree1_cv_results = make_results('decision tree cv', tree1, 'auc')
tree1_cv_results
```

Out[45]:

	model	precision	recall	F1	accuracy	auc
0	decision tree cv	0.914552	0.916949	0.915707	0.971978	0.969819

All of these scores from the decision tree model are strong indicators of good model performance.

constructing a random forest model and setting up cross-validated grid-search to exhuastively search for the best model parameters.

In [46]:

```
rf = RandomForestClassifier(random_state=0)

# Assigning a dictionary of hyperparameters to search over
cv_params = {'max_depth': [3,5, None],
             'max_features': [1.0],
             'max_samples': [0.7, 1.0],
             'min_samples_leaf': [1,2,3],
             'min_samples_split': [2,3,4],
             'n_estimators': [300, 500],
             }

# Assigning a dictionary of scoring metrics to capture
scoring = {'accuracy', 'precision', 'recall', 'f1', 'roc_auc'}

rf1 = GridSearchCV(rf, cv_params, scoring=scoring, cv=4, refit='roc_auc')
```

In [47]:

```
rf1.fit(X_train, y_train)
```

Out[47]:

```
GridSearchCV(cv=4, estimator=RandomForestClassifier(random_state=0),
             param_grid={'max_depth': [3, 5, None], 'max_features': [1.0],
                         'max_samples': [0.7, 1.0],
                         'min_samples_leaf': [1, 2, 3],
                         'min_samples_split': [2, 3, 4],
                         'n_estimators': [300, 500]},
             refit='roc_auc',
             scoring={'f1', 'recall', 'precision', 'accuracy', 'roc_auc'})
```

In [48]:
```
rf1.best_score_
```

Out[48]:
```
0.9804250949807172
```

In [49]:
```
# Checking best params
rf1.best_params_
```

Out[49]:
```
{'max_depth': 5,
 'max_features': 1.0,
 'max_samples': 0.7,
 'min_samples_leaf': 1,
 'min_samples_split': 4,
 'n_estimators': 500}
```

In [50]:
```
# Getting all CV scores
rf1_cv_results = make_results('random forest cv', rf1, 'auc')
print(tree1_cv_results)
print(rf1_cv_results)
```

```
               model  precision    recall        F1  accuracy       auc
0  decision tree cv   0.914552  0.916949  0.915707  0.971978  0.969819
               model  precision    recall        F1  accuracy       auc
0  random forest cv   0.950023  0.915614  0.932467  0.977983  0.980425
```

In [51]:
```
def get_scores(model_name:str, model, X_test_data, y_test_data):
    '''
    Generate a table of test scores.

    In:
        model_name (string):  How you want your model to be named in the output table
        model:                A fit GridSearchCV object
        X_test_data:          numpy array of X_test data
        y_test_data:          numpy array of y_test data

    Out: pandas df of precision, recall, f1, accuracy, and AUC scores for your model
    '''

    preds = model.best_estimator_.predict(X_test_data)

    auc = roc_auc_score(y_test_data, preds)
    accuracy = accuracy_score(y_test_data, preds)
    precision = precision_score(y_test_data, preds)
    recall = recall_score(y_test_data, preds)
    f1 = f1_score(y_test_data, preds)

    table = pd.DataFrame({'model': [model_name],
                          'precision': [precision],
                          'recall': [recall],
                          'f1': [f1],
                          'accuracy': [accuracy],
                          'AUC': [auc]
                         })

    return table
```

In [52]:
```
# Getting predictions on test data
rf1_test_scores = get_scores('random forest1 test', rf1, X_test, y_test)
rf1_test_scores
```

Out[52]:

	model	precision	recall	f1	accuracy	AUC
0	random forest1 test	0.964211	0.919679	0.941418	0.980987	0.956439

Feature Engineering

In [53]:
```python
# Dropping `satisfaction_level` and save resulting dataframe in new variable
df2 = df_enc.drop('satisfaction_level', axis=1)

# Displaying first few rows of new dataframe
df2.head()
```

Out[53]:

	last_evaluation	number_project	average_monthly_hours	tenure	work_accident	left	promotion_last_5y
0	0.53	2	157	3	0	1	
1	0.86	5	262	6	0	1	
2	0.88	7	272	4	0	1	
3	0.87	5	223	5	0	1	
4	0.52	2	159	3	0	1	

In [54]:
```python
# Createing `overworked` column. For now, it's identical to average monthly hours.
df2['overworked'] = df2['average_monthly_hours']

# Inspect max and min average monthly hours values
print('Max hours:', df2['overworked'].max())
print('Min hours:', df2['overworked'].min())
```

```
Max hours: 310
Min hours: 96
```

In [55]:
```python
# Defining `overworked` as working > 175 hrs/week
df2['overworked'] = (df2['overworked'] > 175).astype(int)

# Displaying first few rows of new column
df2['overworked'].head()
```

Out[55]:
```
0    0
1    1
2    1
3    1
4    0
Name: overworked, dtype: int64
```

In [56]:
```python
# Dropping the `average_monthly_hours` column
df2 = df2.drop('average_monthly_hours', axis=1)

# Displaying first few rows of resulting dataframe
df2.head()
```

Out[56]:

	last_evaluation	number_project	tenure	work_accident	left	promotion_last_5years	salary	department
0	0.53	2	3	0	1	0	0	
1	0.86	5	6	0	1	0	1	
2	0.88	7	4	0	1	0	1	
3	0.87	5	5	0	1	0	0	
4	0.52	2	3	0	1	0	0	

In [57]:
```python
# Isolating the outcome variable
y = df2['left']
```

```python
# Selecting the features
X = df2.drop('left', axis=1)
```

In [58]:
```python
# Creating test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, stratify=y, ra
```

In [59]:
```python
tree = DecisionTreeClassifier(random_state=0)

# Assigning a dictionary of hyperparameters to search over
cv_params = {'max_depth':[4, 6, 8, None],
             'min_samples_leaf': [2, 5, 1],
             'min_samples_split': [2, 4, 6]
             }

# Assigning a dictionary of scoring metrics to capture
scoring = {'accuracy', 'precision', 'recall', 'f1', 'roc_auc'}

tree2 = GridSearchCV(tree, cv_params, scoring=scoring, cv=4, refit='roc_auc')
```

In [60]:
```python
tree2.fit(X_train, y_train)
```

Out[60]:
```
GridSearchCV(cv=4, estimator=DecisionTreeClassifier(random_state=0),
             param_grid={'max_depth': [4, 6, 8, None],
                         'min_samples_leaf': [2, 5, 1],
                         'min_samples_split': [2, 4, 6]},
             refit='roc_auc',
             scoring={'f1', 'recall', 'precision', 'accuracy', 'roc_auc'})
```

In [61]:
```python
# Checking best AUC score on CV
tree2.best_score_
```

Out[61]:
```
0.9586752505340426
```

In [62]:
```python
# Get all CV scores
tree2_cv_results = make_results('decision tree2 cv', tree2, 'auc')
print(tree1_cv_results)
print(tree2_cv_results)
```

```
               model  precision    recall        F1  accuracy       auc
0  decision tree cv   0.914552  0.916949  0.915707  0.971978  0.969819
                model  precision    recall        F1  accuracy       auc
0  decision tree2 cv   0.856693  0.903553  0.878882  0.958523  0.958675
```

In [63]:
```python
rf = RandomForestClassifier(random_state=0)

# Assign a dictionary of hyperparameters to search over
cv_params = {'max_depth': [3,5, None],
             'max_features': [1.0],
             'max_samples': [0.7, 1.0],
             'min_samples_leaf': [1,2,3],
             'min_samples_split': [2,3,4],
             'n_estimators': [300, 500],
             }

# Assign a dictionary of scoring metrics to capture
scoring = {'accuracy', 'precision', 'recall', 'f1', 'roc_auc'}


rf2 = GridSearchCV(rf, cv_params, scoring=scoring, cv=4, refit='roc_auc')
```

In [64]:
```python
rf2.fit(X_train, y_train)
```

Out[64]:
```
GridSearchCV(cv=4, estimator=RandomForestClassifier(random_state=0),
             param_grid={'max_depth': [3, 5, None], 'max_features': [1.0],
                         'max_samples': [0.7, 1.0],
                         'min_samples_leaf': [1, 2, 3],
                         'min_samples_split': [2, 3, 4],
                         'n_estimators': [300, 500]},
             refit='roc_auc',
             scoring={'f1', 'recall', 'precision', 'accuracy', 'roc_auc'})
```

In [65]:
```
rf2.best_params_
```

Out[65]:
```
{'max_depth': 5,
 'max_features': 1.0,
 'max_samples': 0.7,
 'min_samples_leaf': 2,
 'min_samples_split': 2,
 'n_estimators': 300}
```

In [66]:
```
rf2.best_score_
```

Out[66]:
```
0.9648100662833985
```

In [67]:
```
# Getting all CV scores
rf2_cv_results = make_results('random forest2 cv', rf2, 'auc')
print(tree2_cv_results)
print(rf2_cv_results)
```

```
               model  precision    recall        F1  accuracy       auc
0  decision tree2 cv   0.856693  0.903553  0.878882  0.958523  0.958675
               model  precision    recall        F1  accuracy      auc
0  random forest2 cv   0.866758  0.878754  0.872407  0.957411  0.96481
```

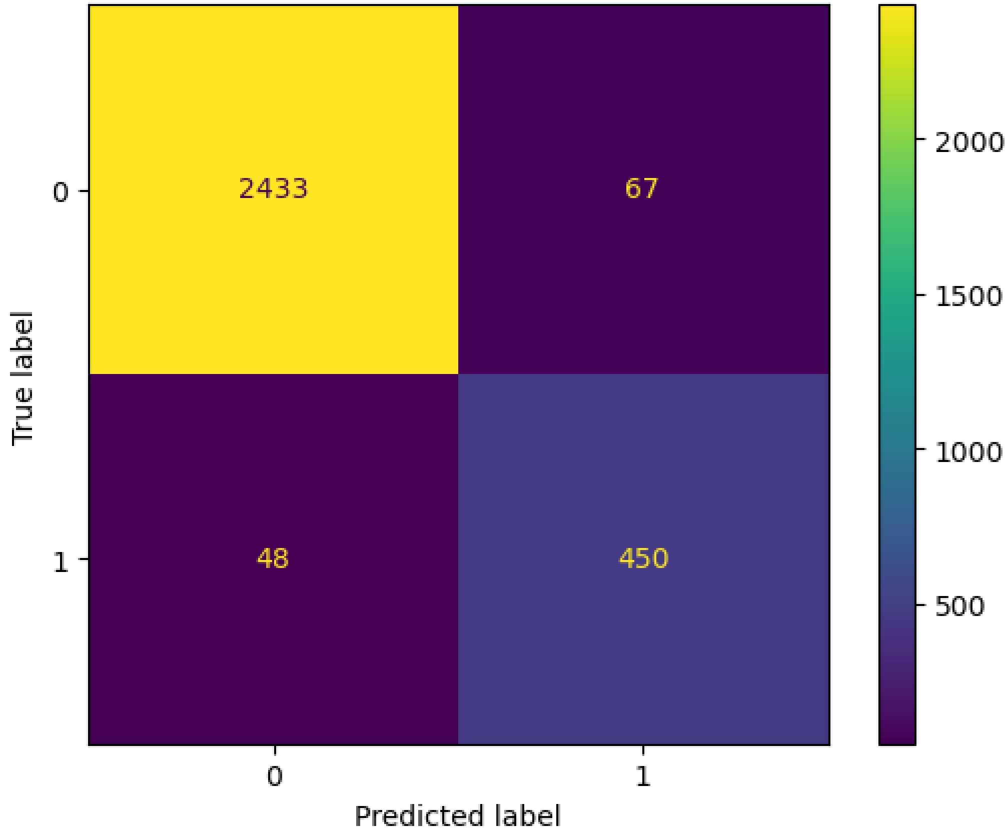
The scores dropped slightly, but the random forest performs better than the decision tree if using AUC as the deciding metric.

In [68]:
```
# Getting predictions on test data
rf2_test_scores = get_scores('random forest2 test', rf2, X_test, y_test)
rf2_test_scores
```

Out[68]:

	model	precision	recall	f1	accuracy	AUC
0	random forest2 test	0.870406	0.903614	0.8867	0.961641	0.938407

In [69]:
```
# Generating array of values for confusion matrix
preds = rf2.best_estimator_.predict(X_test)
cm = confusion_matrix(y_test, preds, labels=rf2.classes_)

# Plotting confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                             display_labels=rf2.classes_)
disp.plot(values_format='');
```



The model predicts more false positives than false negatives, which means that some employees may be identified as at risk of quitting or getting fired, when that's actually not the case. But this is still a strong model.

In [70]:
```
#tree2_importances = pd.DataFrame(tree2.best_estimator_.feature_importances_, columns=X.
tree2_importances = pd.DataFrame(tree2.best_estimator_.feature_importances_,
                                 columns=['gini_importance'],
                                 index=X.columns
                                )
tree2_importances = tree2_importances.sort_values(by='gini_importance', ascending=False)

# Only extract the features with importances > 0
tree2_importances = tree2_importances[tree2_importances['gini_importance'] != 0]
tree2_importances
```

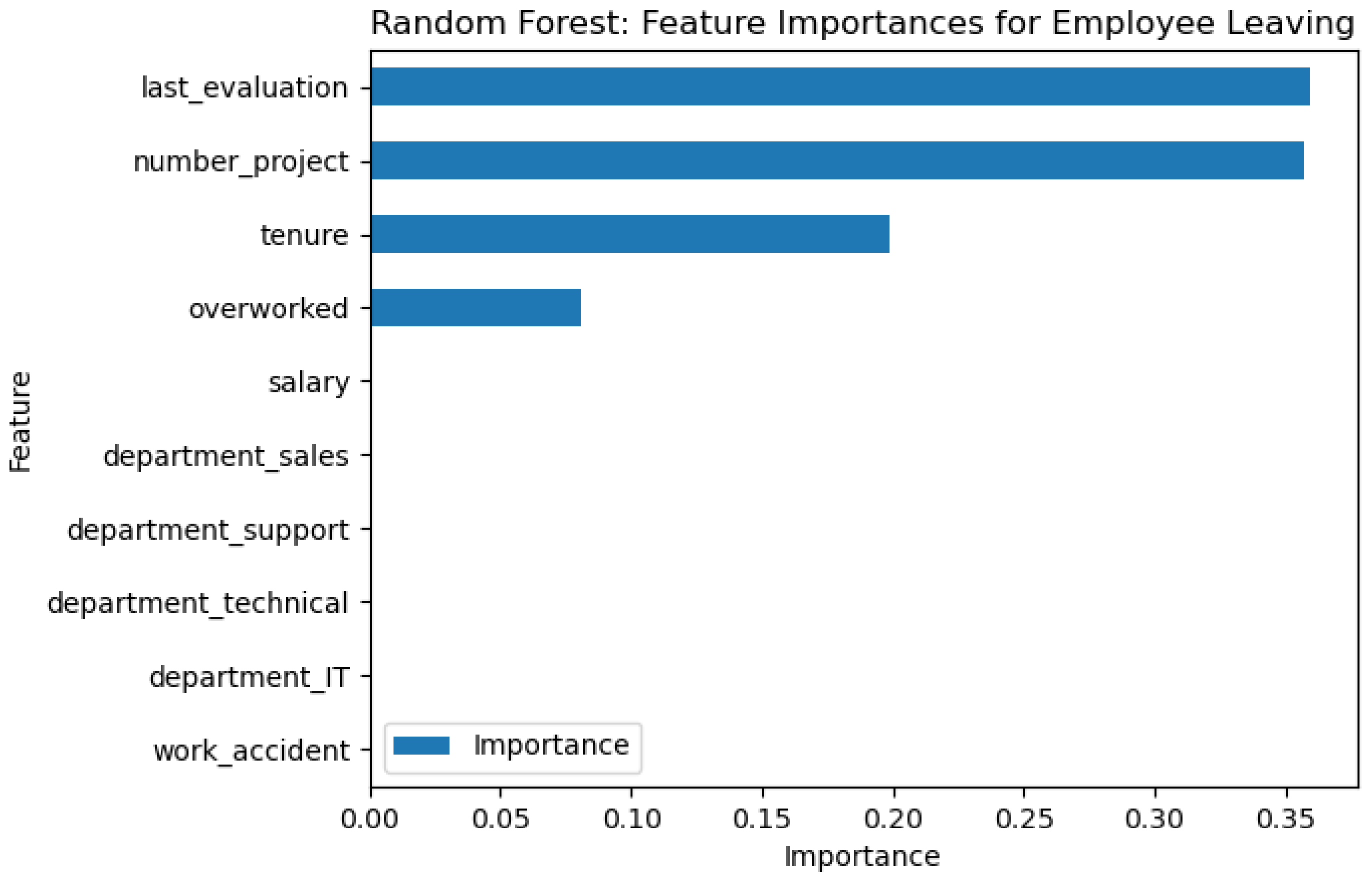
Out[70]:

	gini_importance
last_evaluation	0.343958
number_project	0.343385
tenure	0.215681
overworked	0.093498
department_support	0.001142
salary	0.000910
department_sales	0.000607
department_technical	0.000418
work_accident	0.000183
department_IT	0.000139
department_marketing	0.000078

In [71]:
```
sns.barplot(data=tree2_importances, x="gini_importance", y=tree2_importances.index, orie
plt.title("Decision Tree: Feature Importances for Employee Leaving", fontsize=12)
plt.ylabel("Feature")
plt.xlabel("Importance")
plt.show()
```

The barplot above shows that in this decision tree model, last_evaluation, number_project, tenure, and overworked have the highest importance, in that order.

In [72]:
```
# Get feature importances
feat_impt = rf2.best_estimator_.feature_importances_

# Get indices of top 10 features
ind = np.argpartition(rf2.best_estimator_.feature_importances_, -10)[-10:]

# Get column labels of top 10 features
feat = X.columns[ind]

# Filter `feat_impt` to consist of top 10 feature importances
feat_impt = feat_impt[ind]

y_df = pd.DataFrame({"Feature":feat,"Importance":feat_impt})
y_sort_df = y_df.sort_values("Importance")
fig = plt.figure()
ax1 = fig.add_subplot(111)

y_sort_df.plot(kind='barh',ax=ax1,x="Feature",y="Importance")

ax1.set_title("Random Forest: Feature Importances for Employee Leaving", fontsize=12)
ax1.set_ylabel("Feature")
ax1.set_xlabel("Importance")

plt.show()
```



The plot above shows that in this random forest model, last_evaluation, number_project, tenure, and overworked have the highest importance, in that order. These variables are most helpful in predicting the outcome variable, left, and they are the same as the ones used by the decision tree model.

Summary of model results Logistic Regression

The logistic regression model achieved precision of 80%, recall of 83%, f1-score of 80% (all weighted averages), and accuracy of 83%, on the test set.

Tree-based Machine Learning

After conducting feature engineering, the decision tree model achieved AUC of 93.8%, precision of 87.0%, recall of 90.4%, f1-score of 88.7%, and accuracy of 96.2%, on the test set. The random forest modestly outperformed the decision tree model.

Conclusion, Recommendations, Next Steps The models and the feature importances extracted from the models confirm that employees at the company are overworked.

To retain employees, the following recommendations could be presented to the stakeholders:

Cap the number of projects that employees can work on. Consider promoting employees who have been with the company for atleast four years, or conduct further investigation about why four-year tenured employees are so dissatisfied. Either reward employees for working longer hours, or don't require them to do so. If employees aren't familiar with the company's overtime pay policies, inform them about this. If the expectations around workload and time off aren't explicit, make them clear. Hold company-wide and within-team discussions to understand and address the company work culture, across the board and in specific contexts. High evaluation scores should not be reserved for employees who work 200+ hours per month. Consider a proportionate scale for rewarding employees who contribute more/put in more effort.

```
In [ ]:
```