Predictions with a Logistic Regression Classifier

Now, we would like to use our preprocessed data to build a logistic regression classifier, or Logit model, to help us predict whether or not a given employee will exhibit *excessive absenteeism*, based on information encoded in the predictors we preprocessed.

Let us begin by importing the usual libraries, loading and taking a preliminary look at our preprocessed data:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set('notebook')
from IPython.display import display, Image, SVG, Math

%matplotlib inline

dataset ='absenteeism_data_preprocessed.csv'
raw = pd.read_csv(dataset)
raw.sample(5)
```

	Reason_1	Reason_2	Reason_3	Reason_4	Month Value	Day of the Week	Transportation Expense	Distance to Work	Age
529	1	0	0	0	10	6	248	25	47
418	0	0	0	1	4	2	179	51	38
130	0	0	1	0	1	1	289	36	33
422	0	0	0	1	4	4	260	50	36
484	0	0	0	1	8	1	289	36	33

The Absenteeism Time in Hours column

This is our target column, the variable we will use to predict whether or not absenteeism occurs given our predictors.

```
print('\033[1m' + 'Basic Stats' + '\033[0m')
abstimestats = raw['Absenteeism Time in Hours'].describe()
abstimestats
```

Basic Stats

```
count    700.000000
mean       6.761429
std       12.670082
min        0.000000
25%        2.000000
50%        3.000000
75%        8.000000
max      120.000000
Name: Absenteeism Time in Hours, dtype: float64
```

Creating our targets

Let's create our targets for the logistic regression. We would like to have a binary classification telling us whether or not an employee is excessively absent. Thus, we can transform this column into a classification containing binary values: True if absent and False if not absent.

One way to achieve this is by mapping all values above a certain threshold amount of Absenteeism hours to 1 and the rest to 0.

The median can be used as our threshold as it automatically balances our data into 2 roughly equal classes

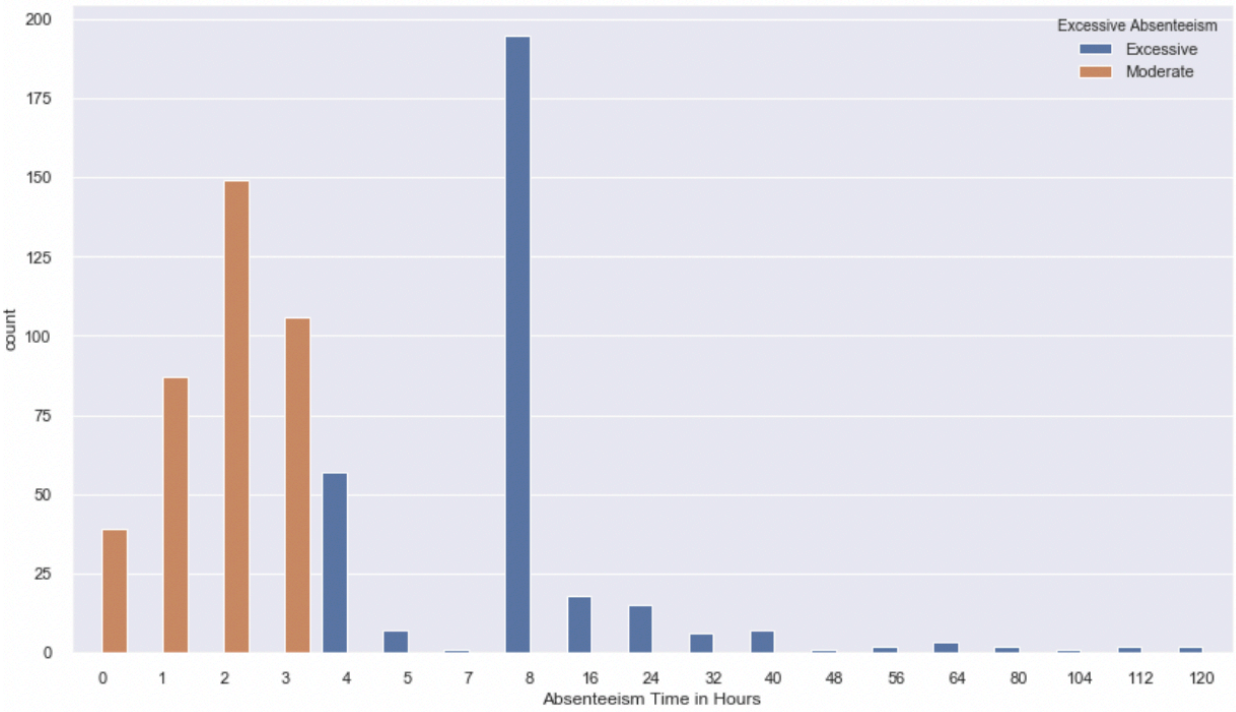
```
med = abstimestats['50%'] # median = 50 percentile!
print('The median is %d'%med)
targets = np.where(raw['Absenteeism Time in Hours']>med, 1, 0)
print('%5.2f%% of the targets are excessively absent. \nA 60/40 split\
 still counts as balanced!' %(100*sum(targets)/np.shape(targets)))
```

```
The median is 3
45.57% of the targets are excessively absent.
A 60/40 split still counts as balanced!
```

```
Targs = pd.DataFrame(data=targets, columns=['Excessive Absenteeism'])
Targs.sample(6)
```

	Excessive Absenteeism
579	1
83	1
311	0
406	0
31	1
213	0

```
plt.rcParams['figure.figsize'] = (14.0, 8.0)
plt.xlabel('Absenteeism in Hours')
sns.countplot(raw['Absenteeism Time in Hours'],
              hue=Targs['Excessive Absenteeism'].map({0:'Moderate', 1:'E
plt.show()
```

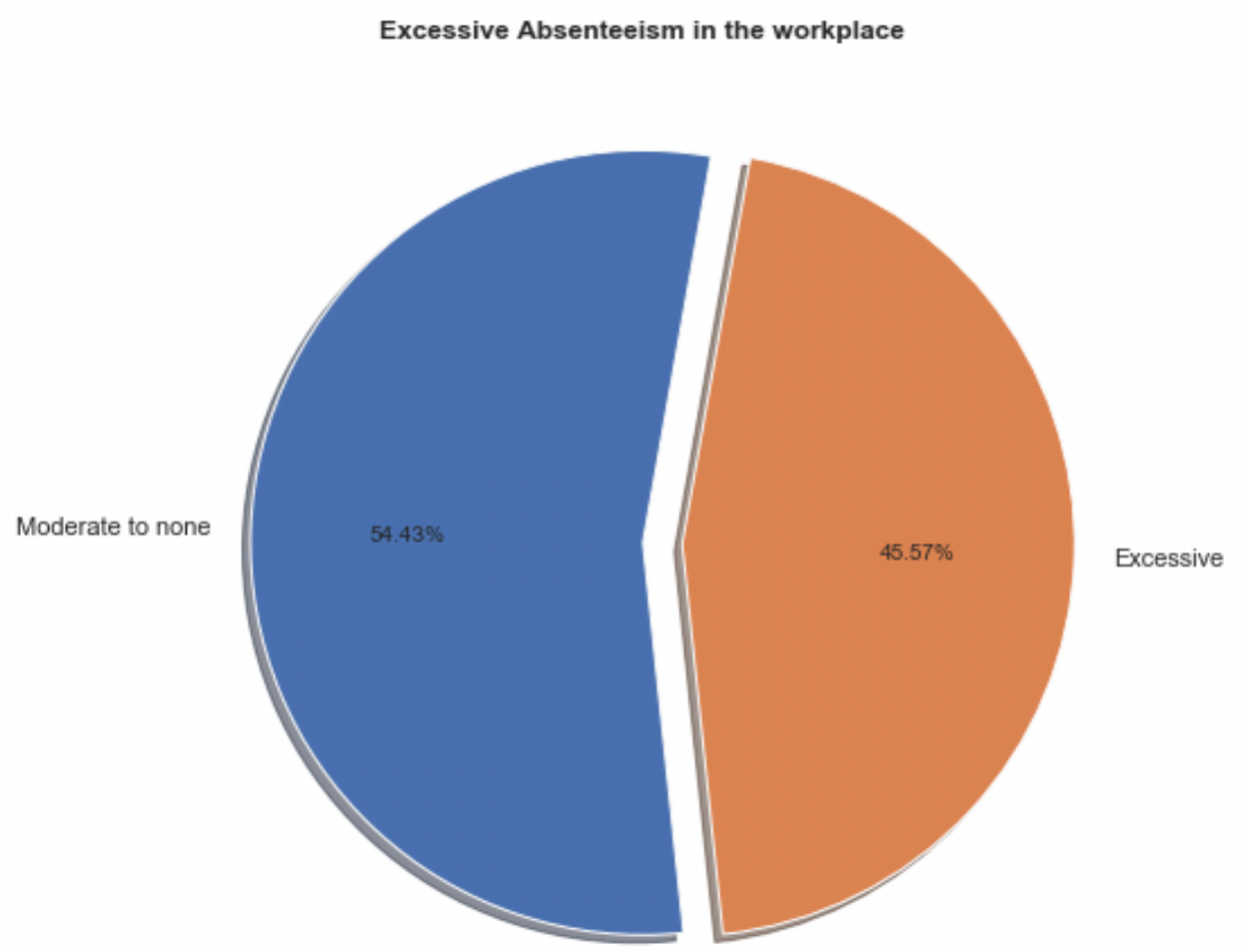


Thus, using logistic regression, we will classify employees into 2 categories:

class 1: Excessively absent ≤*median*≤ **class 2:** moderately to non-absconding.

i.e We've decided to classify an instance where more than 3 hours are taken off work as excessive absenteeism.

```
plt.pie(Targs['Excessive Absenteeism'].value_counts(),  explode=(0,0.1),
        labels=['Moderate to none', 'Excessive'], startangle=80, autopct='%1.2f%%', s

plt.title('Excessive Absenteeism in the workplace', fontweight='bold')
plt.show()
```



Selecting the inputs for regression:

```
raw_inputs = raw.drop(['Absenteeism Time in Hours'], axis=1)
```

we could alternatively use iloc to slice the dataFrame, like so:

```
raw_inputs = df.iloc[:,:-1]
```

where the first argument selects all rows and after the comma we are selecting all columns but the last one.

Train/ Test Split
We divide our dataset into two parts:

- The Training set, on which we will train our model. This chunk is typically 70-80% of the entire dataset.
- The Validation/Test set, also known as the holdout set. This is the remainder of the data, the 'unseen' data on which we will test our model.

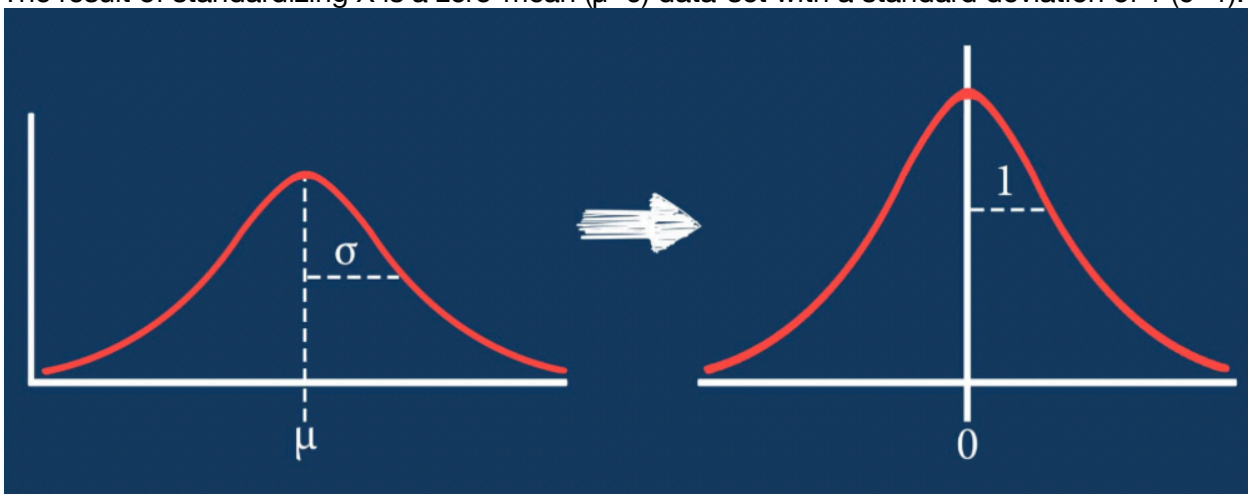
It is always a good idea to perform this step early on, specifically *before* any scaling to prevent any leakage of information between the train and test set.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(raw_inputs, Targs, test_size=0.2
```

Standardizing the data:

Why do we need to standardize our data?
Standardizing allows us to use one distribution (the Normal distribution) when comparing data with different units, ranges or other attributes (i.e multivariate data). It also ensures that the data and results inferred therefrom are comparable with other datasets. Standardization works when the (random) variable, X is normally (N) distributed with mean μ and a variance of σ2:X~N(μ,σ2)

The result of standardizing X is a zero-mean (μ=0) data-set with a standard deviation of 1 (σ=1).



Each observation's standardized score $z_i = \frac{x_i - \mu}{\sigma}$ (a.k.a z-score) tells us, based on the sign of z(±), if the datapoint is below or above the mean, while the magnitude of *z* tells us by how much.

The above two points basically allow us to easily say how far from the *norm* a single observation is, i.e: whether it's 1 σ (read 1 sigma, when z=1) or 3 σ (z=3), etc. above or below

the mean, ultimately allowing us to use a single model to evaluate the likelihood of any observation.

Neglecting to standardize our data can have the effect of skewing the results of a ML algorithm towards, say, a feature with a large range but very little importance to the model in reality!

Standardize
The decision to standardize or not to standardize ultimately depends on the data scientist/ analyst, informed by their ultimate requirements with the data. Some machine learning algorithms' solvers also penalize unscaled data, thus in the interest of accuracy, one may opt to either standardize or create analyses covering both scenarios, which would be simplified by the use of a pipeline.

TLDR: Standardization tells us the average spread of the observations from the mean of the feature, and it is useful for comparing different features and datasets.

Now, we are building a binary classifier, there are some columns which are already binary (0,1) and will not need scaling in our dataset

```
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer

#exclude = [i for i in X_train.columns if np.all(X_train[i].unique() in np.array([0,
exclude = [i for i in X_train.columns if X_train[i].nunique()==2 ]

print('Standardize all inputs except:\n', exclude, '\n\n')

#The columns we WILL be scaling are:

to_scale = [col for col in X_train.columns if col not in exclude]
#print(to_scale, '\n')

coltrans = ColumnTransformer(
      remainder = 'passthrough', #ignore the unspecified columns, without dropping t
      transformers=[('scale', StandardScaler(), to_scale)]
      )

X_train = X_train.astype(float) # StandardScaler expects all data to have dtype=floa
# ColumnTransformer still moves the unscaled columns to the end. We must recover the
X_train_scaled = pd.DataFrame(data=coltrans.fit_transform(X_train), columns=to_scale

print('\033[1m'+'\nOur Predictors, after scaling:\n'+'\033[0m' )
X_train_scaled.head()
```

```
Standardize all inputs except:
 ['Reason_1', 'Reason_2', 'Reason_3', 'Reason_4', 'Education']


*Our Predictors, after scaling:*
```

	Reason_1	Reason_2	Reason_3	Reason_4	Month Value	Day of the Week	Transportation Expense	Distanc to Wor
0	0.0	0.0	0.0	1.0	0.772843	2.682111	-1.563951	-1.37224
1	0.0	0.0	0.0	0.0	0.772843	0.647824	0.178263	-0.69839
2	0.0	0.0	0.0	1.0	1.056269	-0.708368	-0.655617	1.390539
3	0.0	0.0	0.0	1.0	-1.211142	1.325919	-0.655617	1.390539
4	0.0	1.0	0.0	0.0	-0.644289	-0.708368	0.178263	-0.96793

The Logistic Regression

```
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
```

Let's define and train our model

```
logreg = LogisticRegression(solver='newton-cg')
logreg.fit(X_train_scaled, y_train.to_numpy().ravel())
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='newton-cg',
          tol=0.0001, verbose=0, warm_start=False)
```

Predictions and interpretability

Predictions:
Let's see what our model's predictions are, by running model.predict on our holdout set predictors.

NB: Let's not forget to standardize our test set first!

NB: We fit our scaler on the training set only, to teach our model the training set's μ and σ: X_train_scaled= $\frac{X_i - \mu_{train}}{\sigma_{train}}$ The .fit_transform method immediately transforms (scales) our training set after fitting. X_train_scaled = scaler.fit_transform(X_train). Learning MUST only take place within the training set.

After the model has learned the values ofμ and σ from the training set, we use the fitted model to transform the holdout set using: X_test_scaled = $\frac{X_i - \mu_{train}}{\sigma_{train}}$

or X_test_scaled = scaler.transform(X_test) for each column.

y_pred = logreg.predict(X_test_scaled) gives us our model's predictions

```
X_test_scaled = pd.DataFrame(data=coltrans.transform(X_test.astype(float)), columns=t
y_pred = pd.DataFrame(data=(logreg.predict(X_test_scaled)), columns=y_test.columns.va
y_pred.sample(5)
```

	Excessive Absenteeism
138	0
84	1
73	0
16	1
48	1

Model Accuracy

The model accuracy is calculated by comparing the predictions in y_pred with the holdout target column.

Let's see how our model performs on the training and testing predictors respectively. A good model doesn't do significantly worse on the test set, if it does, we've overfit.

The inverse, though, doesn't make much sense. However, a marginal 'improvement' in accuracy is possible due to randomness.

```
print('\033[1m'+'Train Accuracy:'+'\033[0m', logreg.score(X_train_scaled, y_train)) #
print('\033[1m'+'Test Accuracy:'+'\033[0m', logreg.score(X_test_scaled, y_test)) # Tr
# Let's Manually calculate the test accuracy....
manual_test_accuracy = (sum(y_pred.to_numpy()==y_test.to_numpy())/np.shape(y_pred.to_
print('\033[1m'+'Manual Test Accuracy:'+'\033[0m', manual_test_accuracy)
```

```
[**Train Accuracy:** 0.7589285714285714
[**Test Accuracy:** 0.7642857142857142
[**Manual Test Accuracy:** 0.7642857142857142
```

The Weights and Bias

The objective of Regression analysis is to determine the weights (coefficients) and bias, which we then apply to the predictors (inputs) to obtain our predictions (the final result).

$$\gamma = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n + \epsilon$$

where γ is the predicted output

β0 is the bias (intercept in math)

β k, 1≤k≤ n are the weights

```
logreg.coef_ # Each predictor's coefficient
```

```
array([[ 2.6719901 ,  0.46065513,  3.09032814,  0.87137457,  0.07028897,
        -0.17395212,  0.62522164,  0.05633776, -0.1554372 ,  0.0393792 ,
         0.20286036,  0.18514474,  0.49220729, -0.33276522]])
```

```
logreg.intercept_[0] # bias
```

```
-1.7037491273882583
```

Interpretation of the weights and bias

The way these coefficients are displayed here makes it quite hard to match up the inputs to their coefficients.

Also, since we are dealing with a logistic regression model the equation actually looks like this:

$$\gamma = log(odds) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n + \epsilon$$

And since we are interested in finding the odds of excessive absenteeism occurring:

$$odds = e^{log(odds)}$$

Let's create a summary table:

```
summary_table = pd.DataFrame(data=X_train_scaled.columns.values, columns=['Feature na
summary_table['Weight'] = np.transpose(logreg.coef_) # Convert the coefficients into
#display(summary_table)

# To add the intercept to the beginning of the summary table:
summary_table.index +=1 # shift the indices down by one
#display(summary_table) # Space has been created for the intercept to be prepended
summary_table.loc[0] = ['Bias', logreg.intercept_[0]]
summary_table = summary_table.sort_index()

summary_table['Odds_ratio'] = np.exp(summary_table['Weight'])
summary_table['importance'] = [(100*abs(1-abs(i))/(abs(1-abs(summary_table.Odds_ratio
summary_table.sort_values('importance', ascending=False)
```

	Feature name	Weight	Odds_ratio	importance
3	Reason_3	3.090328	21.984291	100.000000
1	Reason_1	2.671990	14.468735	64.184847
4	Reason_4	0.871375	2.390194	6.624928
7	Transportation Expense	0.625222	1.868660	4.139573
0	Bias	-1.703749	0.182000	3.898155
13	Children	0.492207	1.635923	3.030473
2	Reason_2	0.460655	1.585112	2.788334
14	Pets	-0.332765	0.716938	1.348921
11	Body Mass Index	0.202860	1.224901	1.071761
12	Education	0.185145	1.203393	0.969261
6	Day of the Week	-0.173952	0.840337	0.760869
9	Age	-0.155437	0.856041	0.686033
5	Month Value	0.070289	1.072818	0.347013
8	Distance to Work	0.056338	1.057955	0.276183
10	Daily Work Load Average	0.039379	1.040165	0.191404

Now we have our coefficients sorted from most to least important. A weight βk of zero (or close to 0) $\Longrightarrow$ the feature will not impact the model by much. Conversely, a larger weight means that the model depends more heavily on that feature. This is intuitive.

The odds ratio on the other hand: Odds × odds_ratio=new odds i.e if odds are 3:1

and odds_ratio=2 then new odds = 6:1

for a unit change (change of 1)

If odds_ratio = 1, odds = new odds.

Thus, the closer a predictor's odds_ratio is to 1, the lower its significance to the model. We would like to know how important each of our predictors are to the model and its predictions. The weights / Odds ratio offer a crude way of evaluating this. However, for a quick ranking of the predictors, I included an arbitrarily defined *importance* column, with no meaning beyond being a ranking aid. In this column, the importance by the absolute value of each feature's difference with 1, displayed as a percentage of the highest-importance predictor.

Feature Importance provides a much more straightforward and less fluffy way of achieving this…

Feature Importance

The Permutation Importance of a feature is calculated randomly shuffling the feature's rows and checking how much the model's prediction accuracy on the hold out set deteriorates as a result. This is done for each feature without changing any of the other columns. This shuffling and model performance evaluation (by calculating how much the loss function suffers per shuffle) is performed multiple times for each feature, to account for randomness. The final value reported is the average importance weight ± the standard deviation, or the range of variation of the importance weights each time the shuffling is performed.

Now, shuffling a predictor's rows should result in less accurate model predictions, for obvious reasons. Thus, the feature importances are calculated by how negatively shuffling them affects the model.

eli5 does this very thing in no more than 4 lines of code!

```
!pip install eli5
import eli5
from eli5.sklearn import PermutationImportance
perm = PermutationImportance(logreg, random_state=0).fit(X_train_scaled, y_train)

eli5.show_weights(perm, feature_names=X_train_scaled.columns.tolist())
```

Weight	Feature
0.1446 ± 0.0168	Reason_1
0.0771 ± 0.0167	Reason_3
0.0493 ± 0.0080	Transportation Expense
0.0382 ± 0.0062	Children
0.0200 ± 0.0142	Reason_4
0.0132 ± 0.0105	Pets
0.0082 ± 0.0048	Age
0.0025 ± 0.0062	Education
0.0018 ± 0.0078	Daily Work Load Average
0.0018 ± 0.0023	Reason_2
-0.0007 ± 0.0017	Month Value
-0.0007 ± 0.0129	Day of the Week
-0.0011 ± 0.0066	Distance to Work
-0.0029 ± 0.0161	Body Mass Index

Interpretation:
Permutation importances are displayed in descending order.

The weights are reported with an uncertainty denoting how much the base weight varied per shuffle.

Some of the features' reported weights are negative. Does that mean the model's loss function **improved** after random shuffling of the column?

yes and no. Rather, this is a result of happenstance: such features have very low permutation importances and due to random noise, the shuffled data's predictions happen to be slightly more accurate than the unshuffled validation data. This issue is less common with larger datasets, where there is less room for chance. (recall: this dataset has only 700 observations!)

Anyway, we see, from our permutation importances, that Reasons 1 and 3, followed by Transport expense and number of Children have the biggest impact on Absenteeism.

Backward Elimination

is where we simplify our model by removing the low-importance features (with weights ≈0 or odds ratio≈1)

The Day of the Week, Distance to work, Body Mass Index and Month appear to be three such features, according to the Permutation Importance Table.

Let's re-train our model without these features and see if a simplified model performs any better.

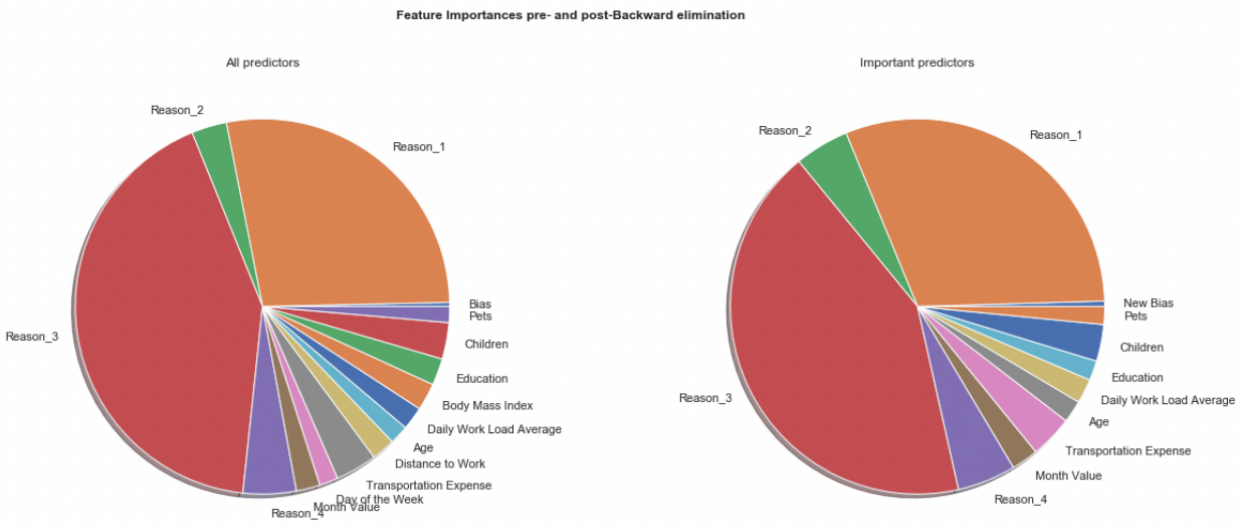
```
# Drop unimportant features from a checkpoint variable. Let's remove variables with
eliminate = ['Distance to Work', 'Month', 'Body Mass Index', 'Day of the Week']
#`Day of the Week`, `Distance to work`, `Body Mass Index` and `Month`
#eliminate = ['Absenteeism Time in Hours','Day of the Week', 'Daily Work Load Averag
feat = raw_inputs.drop([i for i in eliminate if i in raw_inputs], axis=1)

# Train/test split
X_tr, X_te, y_tr, y_te = train_test_split(feat, Targs, test_size=0.2, random_state=0

# Select the columns to standardize
#incl = [i for i in X_tr.columns if np.all(X_tr[i].unique() != np.array([0,1])) and
incl = [i for i in X_tr.columns if X_tr[i].nunique()!=2 ]
excl = [i for i in X_tr.columns if i not in incl]
#standardize
coltransformer = ColumnTransformer(remainder = 'passthrough', transformers=[('scale'
X_tr_scaled = pd.DataFrame(data=coltransformer.fit_transform(X_tr.astype(float)), co
X_te_scaled = pd.DataFrame(data=coltransformer.transform(X_te.astype(float)), columns

### Retrain the model with less features:
logreg_new = LogisticRegression(solver='newton-cg')
logreg_new.fit(X_tr_scaled, y_tr.to_numpy().ravel())
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='newton-cg',
          tol=0.0001, verbose=0, warm_start=False)
```

```
# Summary Table COmparison
summary_table2 = pd.DataFrame(data=X_tr_scaled.columns.values, columns=['Feature name
summary_table2['New Weight'] = np.transpose(logreg_new.coef_) # Convert the coefficie


# To add the intercept to the beginning of the summary table:
summary_table2.index +=1 # shift the indices down by one
# display(summary_table2) # Space has been created for the intercept to be prepended
summary_table2.loc[0] = ['New Bias', logreg_new.intercept_[0]]
summary_table2 = summary_table2.sort_index()
summary_table2['New Odds_ratio'] = np.exp(summary_table2['New Weight'])


summary_table2['New importance'] = [(abs(1-abs(i))/max(abs(1-abs(summary_table2['New


from IPython.display import display_html
def display_side_by_side(*args):
    html_str=''
    for df in args:
        html_str+=df.to_html()
    display_html(html_str.replace('table','table style="display:inline"'),raw=True)

"""
# credit: https://stackoverflow.com/questions/38783027/jupyter-notebook-display-two-p
display_side_by_side(summary_table.sort_values('importance', ascending=False),
                     summary_table2.sort_values('New importance', ascending=False))
"""

perm = PermutationImportance(logreg, random_state=0).fit(X_train_scaled, y_train)
eli5.show_weights(perm, feature_names=X_train_scaled.columns.tolist())

perm2 = PermutationImportance(logreg_new, random_state=0).fit(X_tr_scaled, y_tr)
eli5.show_weights(perm2, feature_names=X_tr.columns.tolist())
```

Weight	Feature
0.1686 ± 0.0262	Reason_1
0.0771 ± 0.0156	Reason_3
0.0236 ± 0.0069	Transportation Expense
0.0225 ± 0.0137	Children
0.0104 ± 0.0089	Reason_4
0.0039 ± 0.0116	Pets
0.0036 ± 0.0045	Reason_2
0.0000 ± 0.0064	Daily Work Load Average
-0.0004 ± 0.0061	Month Value
-0.0014 ± 0.0057	Age
-0.0043 ± 0.0029	Education

```
print('\n----------------------------------------------------------------------------
plt.rcParams['figure.figsize'] = (18.0, 9.0)
f = plt.figure(figsize=(20,8))
f.suptitle('Feature Importances pre- and post-Backward elimination', fontweight='bold
ax = f.add_subplot(121)
ax.set_title('All predictors')
ax.pie(summary_table['Odds_ratio'], labels=summary_table['Feature name'], shadow=True

ax2 = f.add_subplot(122)
ax2.set_title('Important predictors')
ax2.pie(summary_table2['New Odds_ratio'], labels=summary_table2['Feature name'], shad
plt.show()
```



```
trsc = 100*logreg.score(X_train_scaled, y_train)
tsst = 100*logreg.score(X_test_scaled, y_test)
trsc_new = 100*logreg_new.score(X_tr_scaled, y_tr)
tsst_new = 100*logreg_new.score(X_te_scaled, y_te)
print('\033[1m'+'New Accuracies:'+'\033[0m'+'\nTrain Accuracy= %.2f%%\t Test Accuracy
print('\033[1m'+'Old Accuracies:'+'\033[0m'+'\nTrain Accuracy= %.2f%%\t Test Accuracy
      \n ----------------------------------------------------------\n' %(trsc, ts

score_table = pd.DataFrame(data=[trsc, tsst], columns=['old score'])
score_table['new score'] = [trsc_new, tsst_new]
score_table.index = ['train accuracy', 'test accuracy']
score_table
```

```
**New Accuracies:**
Train Accuracy= 74.64%   Test Accuracy = 74.643%
**Old Accuracies:**
Train Accuracy= 75.89%   Test Accuracy = 76.429%
 ----------------------------------------------------------
```

	old score	new score
train accuracy	75.892857	74.642857
test accuracy	76.428571	77.142857

Productionalization the model

We can easily save our final model as a pickle file.

```
import pickle

with open('model', 'wb') as file:
    pickle.dump(logreg_new, file)
```

Let's also save the scaler we used to standardize our data!

```
with open('scaler', 'wb') as file:
    pickle.dump(coltransformer, file)
```

And that's it for now…