# hiring_classification

July 4, 2024

# 1 Supervised Classification: Predicting Hiring Decisions


### 1.0.1 Anthony R. Poggioli


## 1.1 Description

This project was completed as part of the IBM course "Supervised Machine Learning: Classification," which is itself part of the IBM Machine Learning Professional Certificate on Coursera. In this project, I compare the efficacy of several different binary classification machine learning algorithms in predicting if an interviewed candidate is subsequently hired based on 10 independent features described below.

## 1.2 Project Goals

### 1.2.1 Business Goals

The business goal of this project is to determine the factors that are most important in determining if a candidate is hired or not. Specifically, we would like to develop an interpretable ML classifier that will be able to both predict if a candidate is hired based on certain features and explain the most relevant features in determining if a person is hired. Note that these need not be the same model. It may be that the best predictive model is not highly interpretable, in which case an interpretable surrogate model with necessarily inferior predictive performance will also be developed.

In a business context, such a project may be useful for a variety of reasons, including * determining what factors contribute to a successful hiring outcome so that the hiring procedure can be appropriately refocused and made more efficient, and * ensuring that the factors leading to candidate success are aligned with the company's values and strategies.

### 1.2.2 Learning Goals

My goals in completing this project are to deomonstrate 1. proficiency with an array of binary classification algorithms implimented in `scikit-learn`, 2. understanding of cross-validation for hyperparameter tuning, 3. understanding of error metrics appropriate for classification and generalization error estimation, and 4. proficiency with basic `Python` libraries like `NumPy`, `Pandas`, `Matplotlib`, and `seaborn`.

## 1.3 Data

The data used here is taken from the Kaggle dataset Predicting Hiring Decisions in Recruitment Data. I selected this dataset because it represents a relatively simple binary classification problem with a low-dimensional feature space.

Though it is not explicitly specified, this data is likely synthetic. It is also very clean (as I will show below). I therefore will not be able to demonstrate my data cleaning abilities in this project, but these skills are evidenced by other work available on my GitHub – see especially the Google Data Analytics Professional Certificate capstone, cyclistic_case_study.pdf.

### 1.3.1 Features

The following features are included in the data: 1. `Age` (integer) - Age of the candidate in years. 2. `Gender` (integer) - Gender of the candidate, with `0 = Male` and `1 = Female`. 3. `EducationLevel` (integer) - Highest education level attained by the candidate, with `1 = Bachelor's Type 1`, `2 = Bachelor's Type 2`, `3 = Master's`, and `4 = Ph.D.` Note that it is not specified in the description of the data what the difference is between a Type 1 and a Type 2 bachelor's degree. 4. `ExperienceYears` (integer) - Candidate's professional experience in years. 5. `PreviousCompanies` (integer) - The number of companies the candidate has worked for previously. 6. `DistanceFromCompany` (float) - The distance in kilometers from the candidate's residence to the company. 7. `InterviewScore` (integer) - An integer score between 0 and 100 assigned to the candidate based on their interview. 8. `SkillScore` (integer) - An integer score between 0 and 100 assigned to the candidate based on their technical skills. 9. `PersonalityScore` (integer) - An integer score between 0 and 100 assigned to the candidate based on an assesment of their personality traits. 10. `RecruitmentStrategy` (integer) - The strategy adopted by the hiring team, with `1 = Aggressive`, `2 = Moderate`, and `3 = Conservative`. 11. `HiringDecision` (integer) - The outcome of the hiring process with `0 = Not Hired` and `1 = Hired`. This is our target variable.

## 1.4 Data Cleaning and Preliminary Analysis

We begin by familiarizing ourself with the data – including rudimentary statistical analysis and feature selection. We would also perform any necessary data cleaning at this stage, though, as we will see, none is required.

### 1.4.1 Familiarizing Ourselves with the Data

[37]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV, train_test_split,␣
 ↪StratifiedKFold
from sklearn.preprocessing import MinMaxScaler, PolynomialFeatures
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, roc_auc_score

plt.rcParams.update({'font.size': 18})
```

[2]:
```
data = pd.read_csv("recruitment_data.csv")
```

[3]:
```
data.dtypes
```

[3]:
```
Age                       int64
Gender                    int64
EducationLevel            int64
ExperienceYears           int64
PreviousCompanies         int64
DistanceFromCompany     float64
InterviewScore            int64
SkillScore                int64
PersonalityScore          int64
RecruitmentStrategy       int64
HiringDecision            int64
dtype: object
```

All of the data types are consistent with the provided descriptions.

[4]:
```
data.count()
```

[4]:
```
Age                     1500
Gender                  1500
EducationLevel          1500
ExperienceYears         1500
PreviousCompanies       1500
DistanceFromCompany     1500
InterviewScore          1500
SkillScore              1500
PersonalityScore        1500
RecruitmentStrategy     1500
HiringDecision          1500
dtype: int64
```

There are no `null` values present.

[5]:
```
categorical = ["Gender", "EducationLevel", "RecruitmentStrategy",␣
 ↪"HiringDecision"] # numerically encoded categorical variables
non_categorical = [x for x in data.columns if x not in categorical]
data[non_categorical].describe()
```

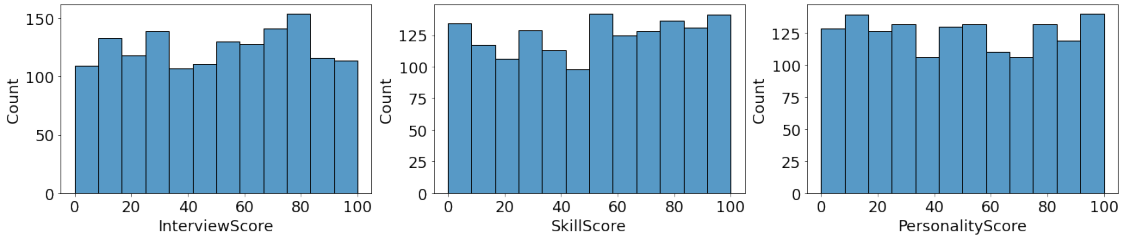
[5]:

| | Age | ExperienceYears | PreviousCompanies | DistanceFromCompany | \ |
|---|---|---|---|---|---|
| count | 1500.000000 | 1500.000000 | 1500.00000 | 1500.000000 | |
| mean | 35.148667 | 7.694000 | 3.00200 | 25.505379 | |
| std | 9.252728 | 4.641414 | 1.41067 | 14.567151 | |
| min | 20.000000 | 0.000000 | 1.00000 | 1.031376 | |
| 25% | 27.000000 | 4.000000 | 2.00000 | 12.838851 | |
| 50% | 35.000000 | 8.000000 | 3.00000 | 25.502239 | |

```
75%      43.000000        12.000000             4.00000            37.737996
max      50.000000        15.000000             5.00000            50.992462

       InterviewScore   SkillScore  PersonalityScore
count     1500.000000  1500.000000       1500.000000
mean        50.564000    51.116000         49.387333
std         28.626215    29.353563         29.353201
min          0.000000     0.000000          0.000000
25%         25.000000    25.750000         23.000000
50%         52.000000    53.000000         49.000000
75%         75.000000    76.000000         76.000000
max        100.000000   100.000000        100.000000
```

All of the non-categorical numerical data is within the specified ranges in the dataset description, and there do not appear to be any outliers. Interestingly, based on the quantile descriptions, the `InterviewScore`, `SkillScore`, and `PersonalityScore` data all appear to be evenly distributed between the minimum (0) and maximum (100) possible values. We can confirm this with histogram plots:

[39]:
```
fig, (ax1, ax2, ax3) = plt.subplots(1,3,figsize = (22,4))
sns.histplot(data["InterviewScore"], ax = ax1)
sns.histplot(data["SkillScore"], ax = ax2)
sns.histplot(data["PersonalityScore"], ax = ax3)
plt.show()
```




Indeed, it appears that the distributions of the scores are uniform. If this were real data, it would likely indicate that these scores have been scaled and possibly curved based on the observed distribution of scores in the sample. If this is synthetic data (the more likely scenario), it indicates that the scores were generated by sampling from a uniform distribution.

[73]:
```
unique_vals = {}
for col in categorical:
    unique_vals[col] = data[col].sort_values().unique().tolist()
print(unique_vals)
```

```
{'Gender': [0, 1], 'EducationLevel': [1, 2, 3, 4], 'RecruitmentStrategy': [1, 2,
3], 'HiringDecision': [0, 1]}
```

For each categorical feature, we see that the values all come from the appropriate set – e.g., all of

the values for `Gender` are either `0` or `1`, and all of the values for `RecruitmentStrategy` are in the set `{1, 2, 3}`.

[81]:
```
hired_fraction = data[data["HiringDecision"] == 1]["HiringDecision"].count()/
 ↪data["HiringDecision"].count()
hired_df = pd.DataFrame({"Decision": ["Not Hired", "Hired"],
                                    "Fraction": [hired_fraction, 1 -␣
 ↪hired_fraction]})
hired_df.head()
```

[81]:

| | Decision | Fraction |
|---|---|---|
| 0 | Not Hired | 0.31 |
| 1 | Hired | 0.69 |

We see that about 31% percent of the data corresponds to a positive hiring decision. This is not an extreme imbalance, and I will account for it only by using `StratifiedKFold` to ensure that the class proportions are preserved during cross-validation.

[83]:
```
male_fraction = data[data["Gender"] == 0]["Gender"].count()/data["Gender"].
 ↪count()
gender_df = pd.DataFrame({"Gender": ["Male", "Female"],
                          "Fraction": [male_fraction, 1 - male_fraction]})
gender_df.head()
```

[83]:

| | Gender | Fraction |
|---|---|---|
| 0 | Male | 0.508 |
| 1 | Female | 0.492 |

The data is approximately evenly split between male and female.

[87]:
```
aggressive_fraction = data[data["RecruitmentStrategy"] ==␣
 ↪1]["RecruitmentStrategy"].count()/data["RecruitmentStrategy"].count()
moderate_fraction = data[data["RecruitmentStrategy"] ==␣
 ↪2]["RecruitmentStrategy"].count()/data["RecruitmentStrategy"].count()

strategy_df = pd.DataFrame({"Strategy": ["Aggressive", "Moderate",␣
 ↪"Conservative"],
                            "Fraction": [aggressive_fraction,␣
 ↪moderate_fraction, 1 - (aggressive_fraction + moderate_fraction)]})
strategy_df.head()
```

[87]:

| | Strategy | Fraction |
|---|---|---|
| 0 | Aggressive | 0.296667 |
| 1 | Moderate | 0.513333 |
| 2 | Conservative | 0.190000 |

The recruitment strategies are also imbalance, with about 30% of the data corresponding to an agressive strategy, 19% to a conservative strategy, and the remaining 51% to a moderate strategy.

[92]:
```
bachelor1_fraction = data[data["EducationLevel"] == 1]["EducationLevel"].
 ↪count()/data["EducationLevel"].count()
bachelor2_fraction = data[data["EducationLevel"] == 2]["EducationLevel"].
 ↪count()/data["EducationLevel"].count()
masters_fraction = data[data["EducationLevel"] == 3]["EducationLevel"].count()/
 ↪data["EducationLevel"].count()

education_df = pd.DataFrame({"Education Level": ["Type 1 Bachelor's", "Type 2␣
 ↪Bachelor's", "Master's", "Ph.D."],
                              "Fraction": [bachelor1_fraction,␣
 ↪bachelor2_fraction, masters_fraction,
                                           1 - (bachelor1_fraction +␣
 ↪bachelor2_fraction + masters_fraction)]})
education_df.head()
```

[92]:

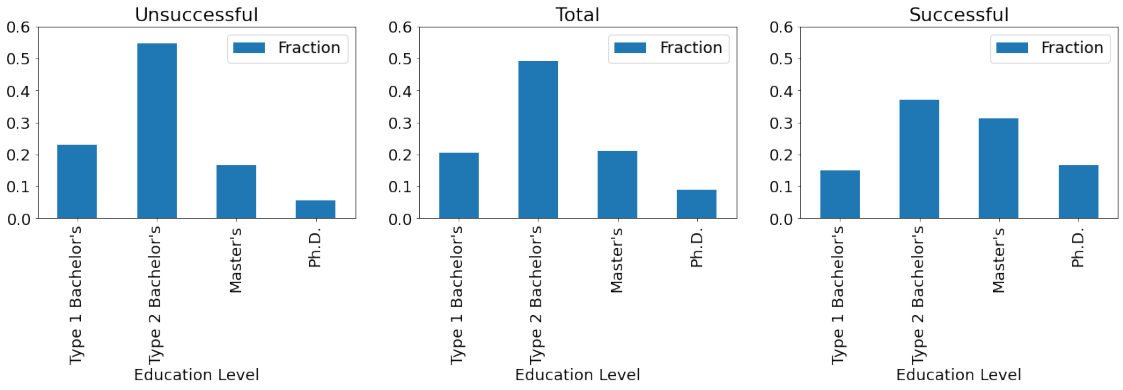
| | Education Level | Fraction |
|---|---|---|
| 0 | Type 1 Bachelor's | 0.204667 |
| 1 | Type 2 Bachelor's | 0.493333 |
| 2 | Master's | 0.211333 |
| 3 | Ph.D. | 0.090667 |

The distribution of education levels is also imbalance, with only about 9% having a Ph.D. and nearly 50% having a type 2 bachelor's degree. Approximately 70% of candidates hold only a bachelor's degree, while the remaining 30% hold a graduate degree. Let's take a look at how these fractions break down for successful and unsuccessful candidates:

[105]:
```
data_u = data[data["HiringDecision"] == 0]
data_s = data[data["HiringDecision"] == 1]

bachelor1_fraction_u = data_u[data_u["EducationLevel"] == 1]["EducationLevel"].
 ↪count()/data_u["EducationLevel"].count()
bachelor2_fraction_u = data_u[data_u["EducationLevel"] == 2]["EducationLevel"].
 ↪count()/data_u["EducationLevel"].count()
masters_fraction_u = data_u[data_u["EducationLevel"] == 3]["EducationLevel"].
 ↪count()/data_u["EducationLevel"].count()

bachelor1_fraction_s = data_s[data_s["EducationLevel"] == 1]["EducationLevel"].
 ↪count()/data_s["EducationLevel"].count()
bachelor2_fraction_s = data_s[data_s["EducationLevel"] == 2]["EducationLevel"].
 ↪count()/data_s["EducationLevel"].count()
masters_fraction_s = data_s[data_s["EducationLevel"] == 3]["EducationLevel"].
 ↪count()/data_s["EducationLevel"].count()

education_u_df = pd.DataFrame({"Education Level": ["Type 1 Bachelor's", "Type 2␣
 ↪Bachelor's", "Master's", "Ph.D."],
                              "Fraction": [bachelor1_fraction_u,␣
 ↪bachelor2_fraction_u, masters_fraction_u,
```

```
                                          1 - (bachelor1_fraction_u +␣
 ↪bachelor2_fraction_u + masters_fraction_u)]})
education_s_df = pd.DataFrame({"Education Level": ["Type 1 Bachelor's", "Type 2␣
 ↪Bachelor's", "Master's", "Ph.D."],
                               "Fraction": [bachelor1_fraction_s,␣
 ↪bachelor2_fraction_s, masters_fraction_s,
                                          1 - (bachelor1_fraction_s +␣
 ↪bachelor2_fraction_s + masters_fraction_s)]})

fig, (ax1, ax2, ax3) = plt.subplots(1,3,figsize = (22,4))
education_u_df.plot(x = "Education Level", y = "Fraction", ax = ax1, kind =␣
 ↪"bar")
education_df.plot(x = "Education Level", y = "Fraction", ax = ax2, kind = "bar")
education_s_df.plot(x = "Education Level", y = "Fraction", ax = ax3, kind =␣
 ↪"bar")
ax1.set_ylim(0,0.6)
ax2.set_ylim(0,0.6)
ax3.set_ylim(0,0.6)
ax1.set_title("Unsuccessful")
ax2.set_title("Total")
ax3.set_title("Successful")
```

[105]: `Text(0.5, 1.0, 'Successful')`




We see that the the relative proportion of bachelor's degrees has decreased for successful candidates, and that this is compensated for by an increase in the relative proportions of advanced degrees. This indicates a preference for advanced degrees in the hiring process, though clearly advanced degrees are not a prerequisite for hiring.

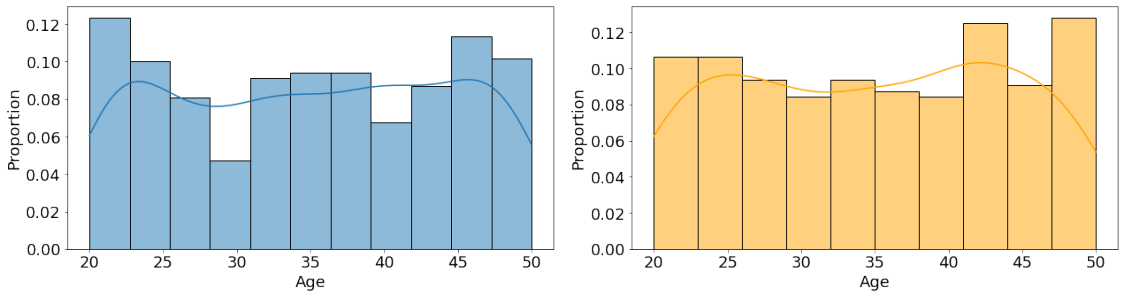
### 1.4.2 Feature Selection

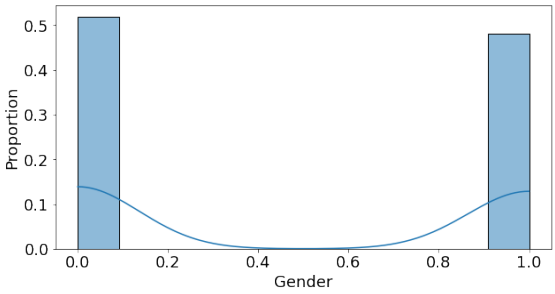
For the sake of interpretability, we will use supervised feature selection to eliminate features that are weakly correlated or uncorrelated with the target.

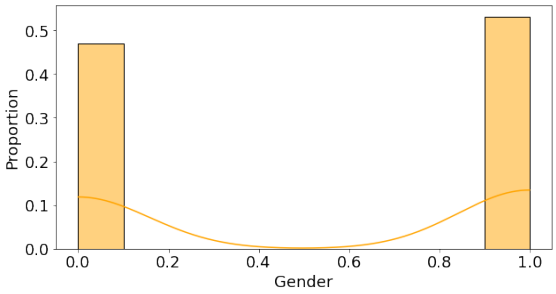
A more careful procedure would be to select the relevant features at each step in cross-validation. Here, we will select features based on their correlation with the target variable *in our training sample*. This ensures that our estimate of the generalization error using the test sample will account for statistical variation in feature importance.

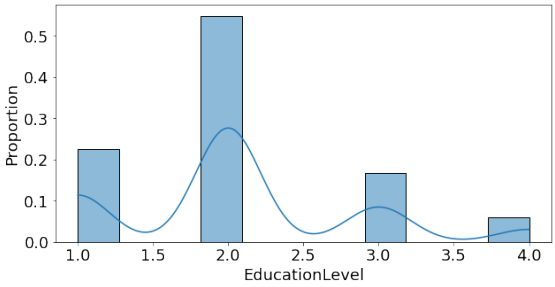
[9]:
```
data_train, data_test = train_test_split(data, test_size = 500, random_state =␣
↪42)
y_train = data_train["HiringDecision"]
y_test = data_test["HiringDecision"]
# note that the random seed is specified to ensure that the results are␣
↪reproduced if this Jupyter notebook is rerun
```

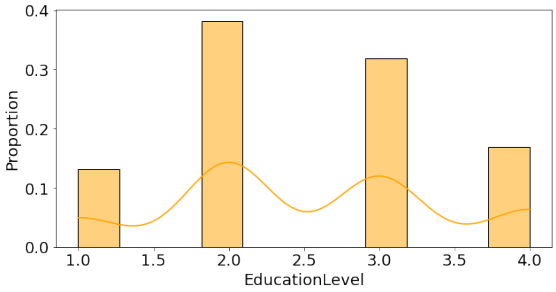
[150]:
```
features = [x for x in data.columns if x != "HiringDecision"]
savename = ["hist_age", "hist_gender", "hist_education", "hist_experience",␣
↪"hist_previous",
            "hist_distance", "hist_interview", "hist_skill",␣
↪"hist_personality", "hist_recruitment"]
count = 0
for feature in features:
    fig, (ax1, ax2) = plt.subplots(1,2,figsize = (18,5))
    sns.histplot(data = data_train[data_train["HiringDecision"] == 0], x =␣
↪feature, stat = "proportion", kde = True, ax = ax1)
    sns.histplot(data = data_train[data_train["HiringDecision"] == 1], x =␣
↪feature, stat = "proportion", color = "orange", kde = True, ax = ax2)
    plt.tight_layout()
    plt.savefig(''.join(("./figures/",savename[count],".pdf")))
    count += 1
    plt.show()
```

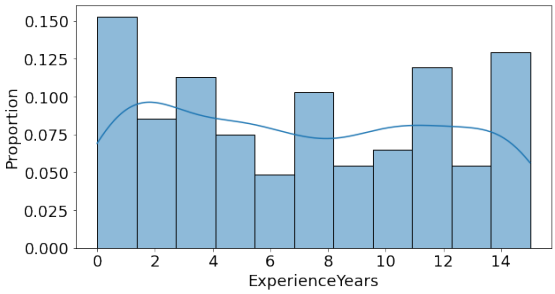


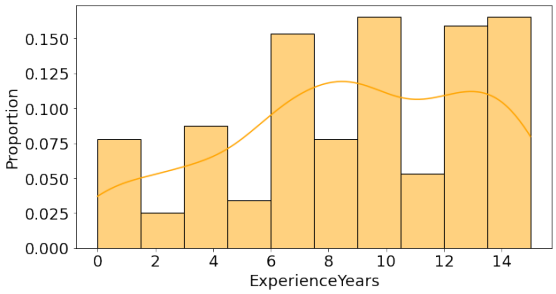

Proportion
Gender

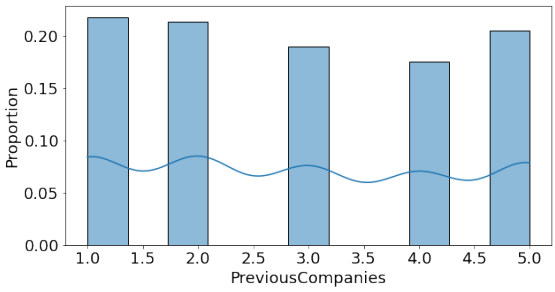

Proportion
Gender

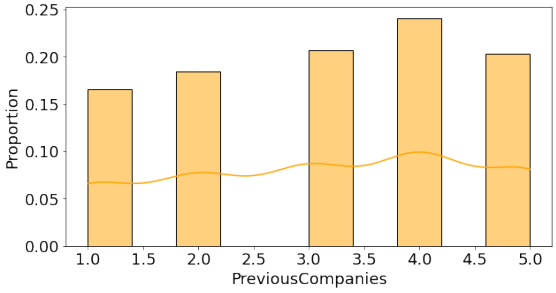

Proportion
EducationLevel

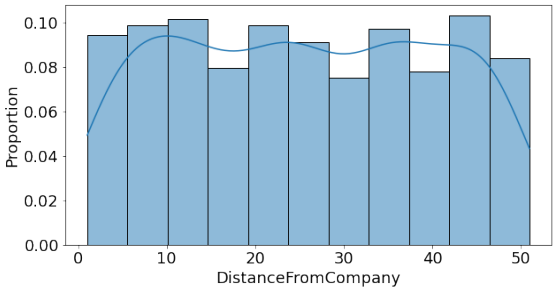

Proportion
EducationLevel

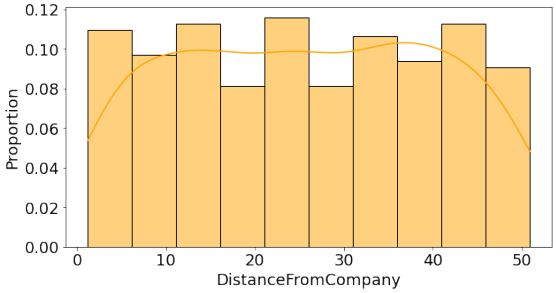

Proportion
ExperienceYears

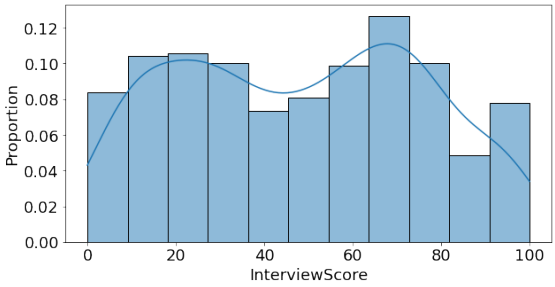

Proportion
ExperienceYears

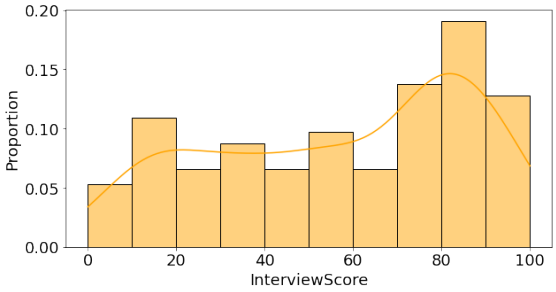

Proportion
PreviousCompanies

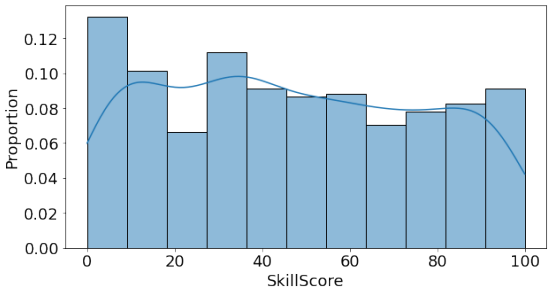

Proportion
PreviousCompanies

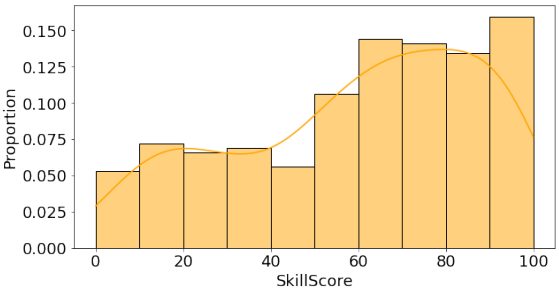

Proportion
0.00
0.02
0.04
0.06
0.08
0.10
0
10
20
30
40
50
DistanceFromCompany

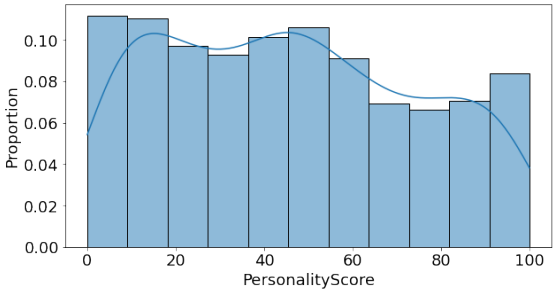

Proportion
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0
10
20
30
40
50
DistanceFromCompany

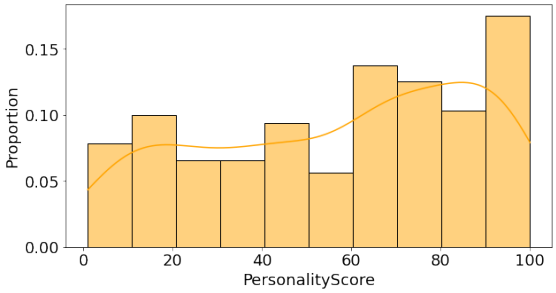

Proportion
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0
20
40
60
80
100
InterviewScore

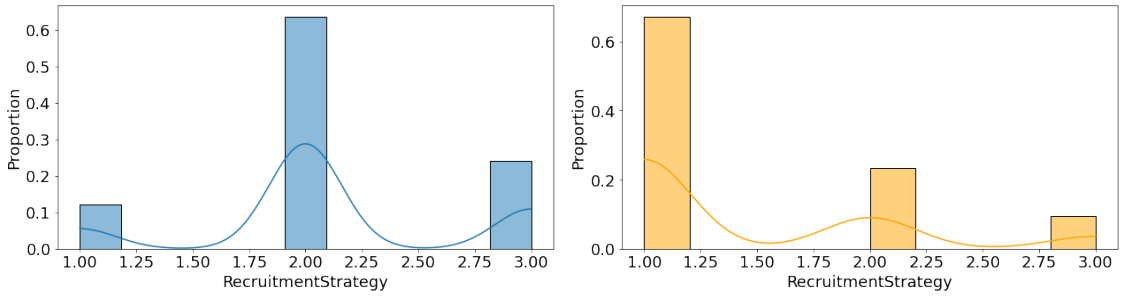

Proportion
0.00
0.05
0.10
0.15
0.20
0
20
40
60
80
100
InterviewScore


Proportion
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0
20
40
60
80
100
SkillScore


Proportion
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0
20
40
60
80
100
SkillScore

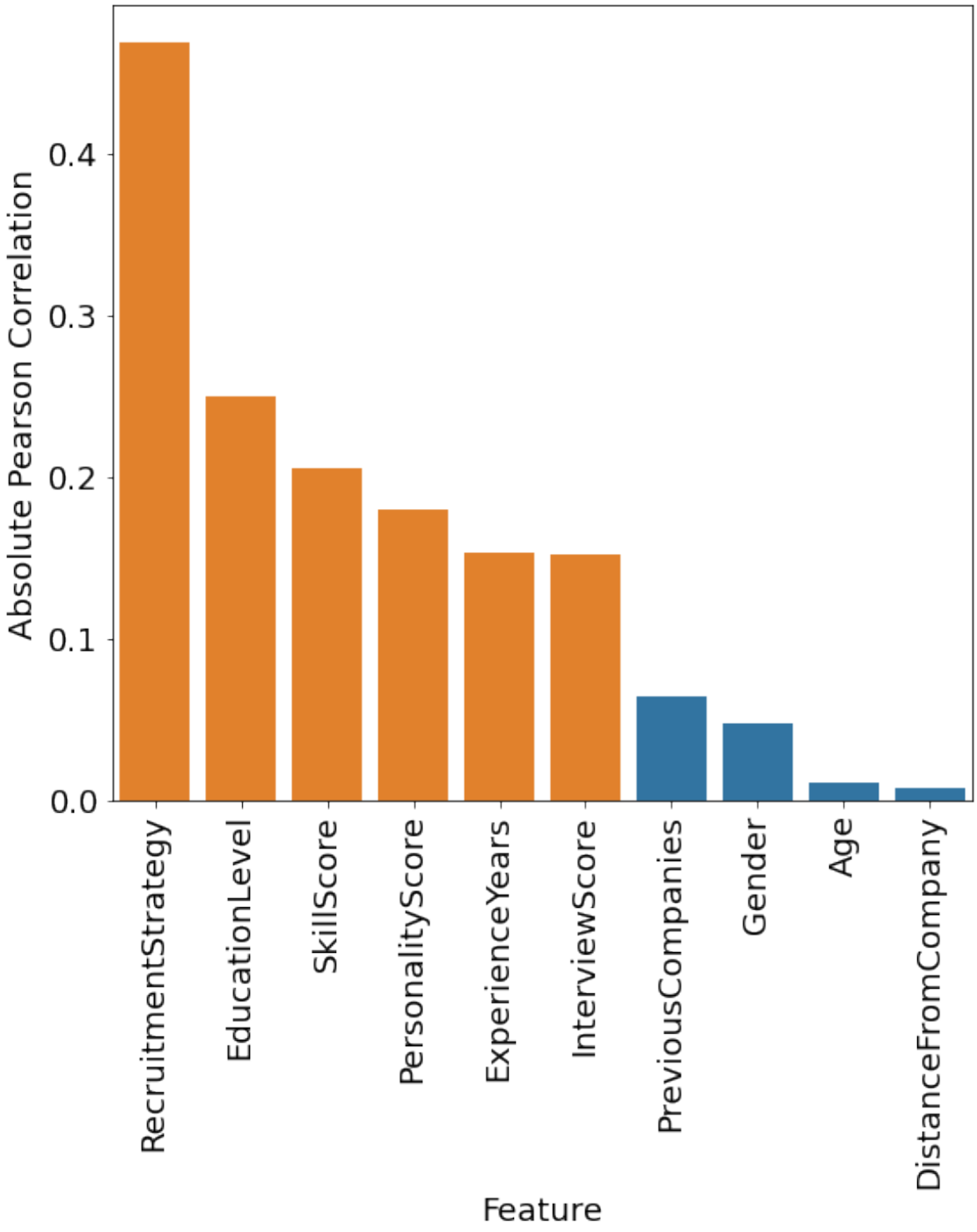

Proportion
0.00
0.02
0.04
0.06
0.08
0.10
0
20
40
60
80
100
PersonalityScore


Proportion
0.00
0.05
0.10
0.15
0
20
40
60
80
100
PersonalityScore




From these histogramss, we see that the distributions of `Age`, `Gender`, `PreviousCompanies`, and `DistanceFromCompany` have little or no correlation with `HiringDecision`, while the remaining features, `EducationLevel`, `ExperienceYears`, `InterviewScore`, `SkillScore`, `PersonalityScore`, and `RecruitmentStrategy` do. We can further confirm this by looking at the correlation of each independent feature with our target:

[132]:
```
ind_features = [x for x in data_train.columns if x != "HiringDecision"]
corr_vec = np.abs(data_train.corr().to_numpy()[-1,:-1])
indices = np.argsort(-corr_vec)
corr_df = pd.DataFrame({"Feature": [ind_features[i] for i in indices],
                        "Absolute Pearson Correlation": corr_vec[indices].
 ↪tolist(),
                        "Relevance": [1, 1, 1, 1, 1, 1, 0, 0, 0, 0]})

fig, ax = plt.subplots(1, figsize = (8,10))
sns.barplot(data = corr_df, x = "Feature", y = "Absolute Pearson Correlation",␣
 ↪ax = ax)
sns.barplot(data = corr_df[corr_df["Relevance"] == 1], x = "Feature", y =␣
 ↪"Absolute Pearson Correlation", ax = ax)
plt.xticks(rotation = "vertical")
plt.tight_layout()
plt.savefig("./figures/feature_correlations.pdf")
plt.show()
```




We see that there is a steep drop-off in the absolute value of the Pearson correlation after `RecruitmentStrategy`, followed by a steady decline and then another steep drop-off after `InterviewScore`. This is consistent with what we have observed above. Based on this, we will eliminate the apparently irrelevant features:

[12]:
```
unimportant_features = ["PreviousCcompanies", "Age", "Gender",␣
↪"DistanceFromCompany"]
selected_features = [x for x in ind_features if x not in unimportant_features]
X_train = data_train[selected_features]
X_test = data_test[selected_features]
```

## 1.5 Models

In this section, I will examine four different binary classification algorithms – (polynomial) logistic regression, k-nearest neighbors, support vector classification, and a decision tree. I will tune the hyperparamters of each of these models using 5-fold grid search cross-validation with stratified folds, and I will select the optimal model based on F1-score. I chose the F1-score because it favors the positive class – in this case successful candidates – but it is balanced in its treatment of false positives and false negatives. This is appropriate for our purposes because we are most interested in identifying successful candidates, but we want to avoid both false positives – alloting resources to interview candidates that will not be successful – and false negatives – failing to identify promising candidates.

As a final note, I will use min-max scaling because several of the features are categorical and/or non-normal, and a standard scaling would not collapse such features to a similar range of values.

[13]:
```
scoring_metric = "f1" # scoring metric used in grid search cross-validation
Nfolds = 5 # number of folds to use in stratified k-fold cross-validation
models = ["LR", "KNN", "SVC", "DT"]
```

### 1.5.1 Logistic Regression

We start with logistic regression. We will imposed L2 (Euclidean/Minkowski)-norm regularization and use grid search cross-validation to tune the regularization parameter. The type of the regularization penalty could aslo be tuned using `GridSearchCV`.

We will also apply a polynomial transformation to our feature vector and use grid search cross-validation to determine the optimal degree of polynomial. We will limit the polynomial degree to be no higher than 3 to help mitigate overfitting.

[14]:
```
lr_pipe = Pipeline([("poly", PolynomialFeatures(include_bias = False)),
                    ("scaler", MinMaxScaler()),
                    ("logreg", LogisticRegression(penalty = "l2", max_iter =␣
↪1000, random_state = 42))])
parameters = {"poly__degree": [x for x in range(1,4)],
              "logreg__C": np.logspace(-4,4,50).tolist()}
lr_GSCV = GridSearchCV(lr_pipe, param_grid = parameters, n_jobs = -1, cv =␣
↪StratifiedKFold(n_splits = Nfolds), scoring = scoring_metric)
lr_GSCV.fit(X_train, y_train)
y_hat_lr = lr_GSCV.predict(X_test)
lr_score_df = pd.DataFrame({"Metric": ["accuracy", "precision", "recall",␣
↪"f1-score"],
                            "Score": [accuracy_score(y_test, y_hat_lr),␣
↪precision_score(y_test, y_hat_lr), recall_score(y_test, y_hat_lr),
```

```
                                        f1_score(y_test, y_hat_lr)]})
degree_opt = lr_GSCV.best_params_["poly__degree"]
C_opt = lr_GSCV.best_params_["logreg__C"]
print("The optimal polynomial degree is {}.".format(degree_opt))
print("The optimal value of the regularization parameter C = {}. \n".
 ↪format(C_opt))
lr_score_df.head()
```

```
The optimal polynomial degree is 3.
The optimal value of the regularization parameter C = 719.6856730011514.
```

[14]:

| | Metric | Score |
|---|---|---|
| 0 | accuracy | 0.900000 |
| 1 | precision | 0.846715 |
| 2 | recall | 0.800000 |
| 3 | f1-score | 0.822695 |

### 1.5.2 K Nearest Neighbors

We next fit a k-nearest neighbors model. I will tune both the distance metric (L1/Manhattan or L2/Euclidean/Minkowski) and the number of neighbors $k$ using grid search cross-validation.

[15]:
```
knn_pipe = Pipeline([("scaler", MinMaxScaler()),
                     ("knn", KNeighborsClassifier())])
parameters = {"knn__n_neighbors": [i for i in range(1,16)],
              "knn__p": [1, 2]}
knn_GSCV = GridSearchCV(knn_pipe, param_grid = parameters, n_jobs = -1, cv =␣
 ↪StratifiedKFold(n_splits = Nfolds), scoring = scoring_metric)
knn_GSCV.fit(X_train, y_train)
y_hat_knn = knn_GSCV.predict(X_test)
knn_score_df = pd.DataFrame({"Metric": ["accuracy", "precision", "recall",␣
 ↪"f1-score"],
                             "Score": [accuracy_score(y_test, y_hat_knn),␣
 ↪precision_score(y_test, y_hat_knn), recall_score(y_test, y_hat_knn),
                                       f1_score(y_test, y_hat_knn)]})
p_opt = knn_GSCV.best_params_["knn__p"]
n_neighbors_opt = knn_GSCV.best_params_["knn__n_neighbors"]
print("The optimal Lp norm is p = {}.".format(p_opt))
print("The optimal number of neighbors is k = {}. \n".format(n_neighbors_opt))
knn_score_df.head()
```

```
The optimal Lp norm is p = 1.
The optimal number of neighbors is k = 3.
```

[15]:

| | Metric | Score |
|---|---|---|
| 0 | accuracy | 0.894000 |

```
1  precision  0.810811
2     recall  0.827586
3   f1-score  0.819113
```

We find that the Manhattan distance performs better than the Euclidean/Minkowski distance, and the optimal value of $k = 3$. The performance of this model is very similar to the logistic regression model. It has a somewhat better recall, but it performs slightly worse on all of the other evaluated metrics – including accuracy and F1-score.

### 1.5.3 Support Vector Classifier

We next fit a support vector classifier with (Gaussian) radial basis functions, tuning the L2-norm regularization parameter $C$ and the kernel width $\gamma$ using grid search cross-validation.

[144]:
```
svc_pipe = Pipeline([("scaler", MinMaxScaler()),
                     ("svc", SVC(max_iter = 1000000, kernel = "rbf",␣
 ↪random_state = 42))])
parameters = {"svc__C": np.logspace(-4,4,50),
              "svc__gamma": np.logspace(-1,1,50)}
svc_GSCV = GridSearchCV(svc_pipe, param_grid = parameters, n_jobs = -1, cv =␣
 ↪StratifiedKFold(n_splits = Nfolds), scoring = scoring_metric)
svc_GSCV.fit(X_train, y_train)
y_hat_svc = svc_GSCV.predict(X_test)
svc_score_df = pd.DataFrame({"Metric": ["accuracy", "precision", "recall",␣
 ↪"f1-score"],
                             "Score": [accuracy_score(y_test, y_hat_svc),␣
 ↪precision_score(y_test, y_hat_svc), recall_score(y_test, y_hat_svc),
                                       f1_score(y_test, y_hat_svc)]})
C_opt = svc_GSCV.best_params_["svc__C"]
gamma_opt = svc_GSCV.best_params_["svc__gamma"]
print("The optimal regularization parameter is C = {}.".format(C_opt))
print("The optimal kernel width is gamma = {}. \n".format(gamma_opt))
svc_score_df.head()
```

```
The optimal regularization parameter is C = 1.7575106248547894.
The optimal kernel width is gamma = 5.17947467923121.
```

[144]:
```
       Metric     Score
0    accuracy  0.902000
1   precision  0.847826
2      recall  0.806897
3    f1-score  0.826855
```

The performance of the SVC model is similar to the logistic regression and k-nearest neighbors models.

### 1.5.4 Decision Tree

We finally fit a decision tree model to the data, tuning the splitting criterion (Gini coefficient or information gain), the maximum depth of the tree, and the maximum number of features to consider at each split.

[151]:
```
parameters = {"criterion": ["gini", "entropy"],
              "max_depth": [x for x in range(5,20)],
              "max_features": [x for x in range(2,len(selected_features))]}
dt_GSCV = GridSearchCV(DecisionTreeClassifier(random_state = 42),
                       param_grid = parameters)
dt_GSCV.fit(X_train, y_train)
y_hat_dt = dt_GSCV.predict(X_test)
dt_score_df = pd.DataFrame({"Metric": ["accuracy", "precision", "recall",␣
 ↪"f1-score"],
                            "Score": [accuracy_score(y_test, y_hat_dt),␣
 ↪precision_score(y_test, y_hat_dt), recall_score(y_test, y_hat_dt),
                                      f1_score(y_test, y_hat_dt)]})
criterion_opt = dt_GSCV.best_params_["criterion"]
max_depth_opt = dt_GSCV.best_params_["max_depth"]
max_features_opt = dt_GSCV.best_params_["max_features"]
print("The optimal criterion is: {}.".format(criterion_opt))
print("The optimal maximum depth is {}.".format(max_depth_opt))
print("The optimal number of features to consider at a split is {}. \n".
 ↪format(max_features_opt))
dt_score_df.head()
```

```
The optimal criterion is: entropy.
The optimal maximum depth is 6.
The optimal number of features to consider at a split is 5.
```

[151]:

| | Metric | Score |
|---|---|---|
| 0 | accuracy | 0.912000 |
| 1 | precision | 0.897638 |
| 2 | recall | 0.786207 |
| 3 | f1-score | 0.838235 |

The performance of the optimal decision tree slightly improves over the performance of the previous models.

## 1.6 Identifying the Best Model

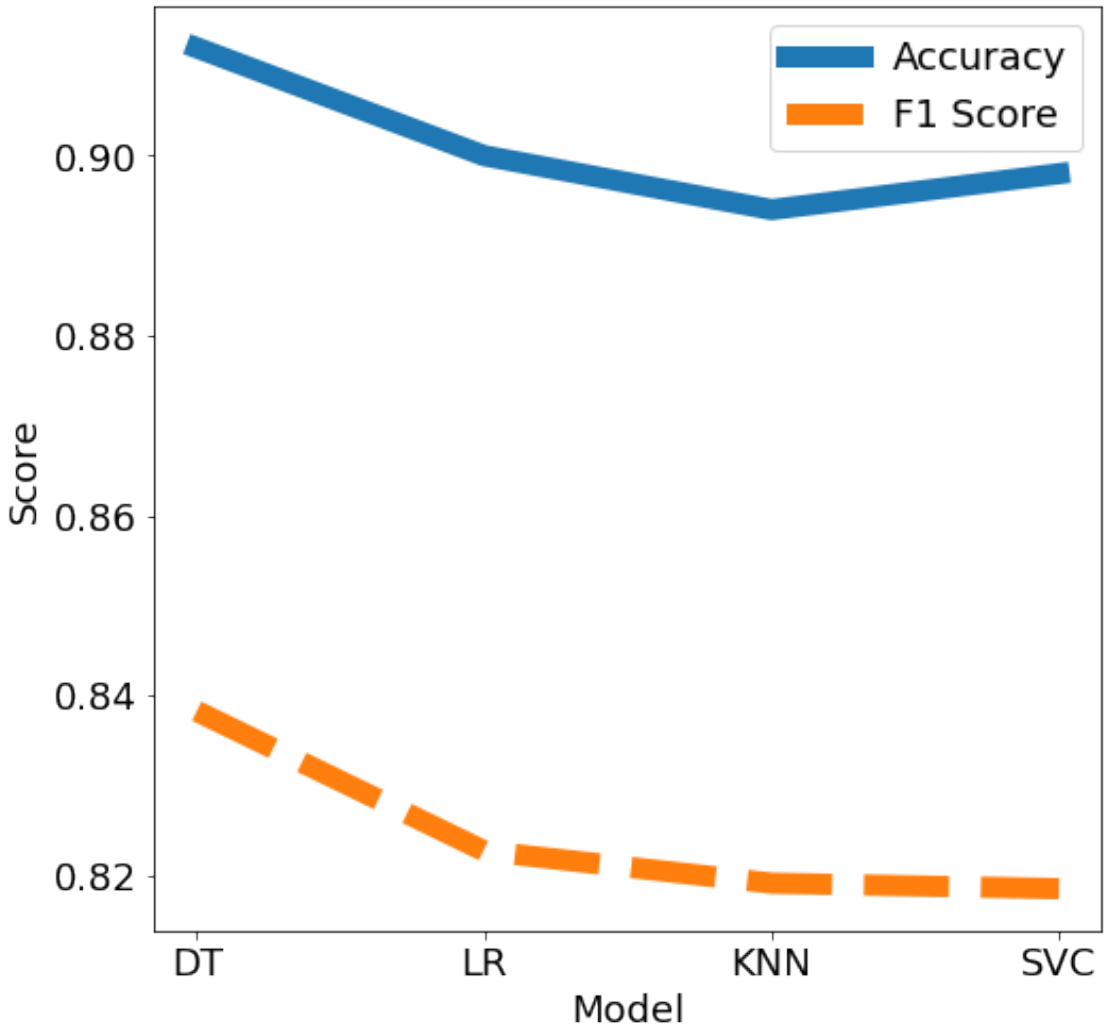
[18]:
```
accuracy_vec = [lr_score_df.iloc[0,1], knn_score_df.iloc[0,1], svc_score_df.
 ↪iloc[0,1], dt_score_df.iloc[0,1]]
f1score_vec = [lr_score_df.iloc[3,1], knn_score_df.iloc[3,1], svc_score_df.
 ↪iloc[3,1], dt_score_df.iloc[3,1]]

indices = np.argsort(-np.array(f1score_vec)).tolist()
```

```
sorted_accuracy = [accuracy_vec[i] for i in indices]
sorted_f1 = [f1score_vec[i] for i in indices]
sorted_models = [models[i] for i in indices]

score_df = pd.DataFrame({"Models": sorted_models,
                         "Accuracy": sorted_accuracy,
                         "F1 Score": sorted_f1})
score_df.head()
```

[18]:

| | Models | Accuracy | F1 Score |
|---|---|---|---|
| 0 | DT | 0.912 | 0.838235 |
| 1 | LR | 0.900 | 0.822695 |
| 2 | KNN | 0.894 | 0.819113 |
| 3 | SVC | 0.898 | 0.818505 |

[161]:
```
fig, ax = plt.subplots(1, figsize = (8,8))
sns.lineplot(data = score_df, ax = ax, lw = 10)
ax.set_xlabel("Model")
ax.set_ylabel("Score")
ax.set_xticks([0, 1, 2, 3])
ax.set_xticklabels(sorted_models)
plt.savefig("./figures/model_evaluation.pdf")
plt.show()
```




We see that the decision tree model performs (slightly) better than the other models considered in terms of both F1-score and accuracy.

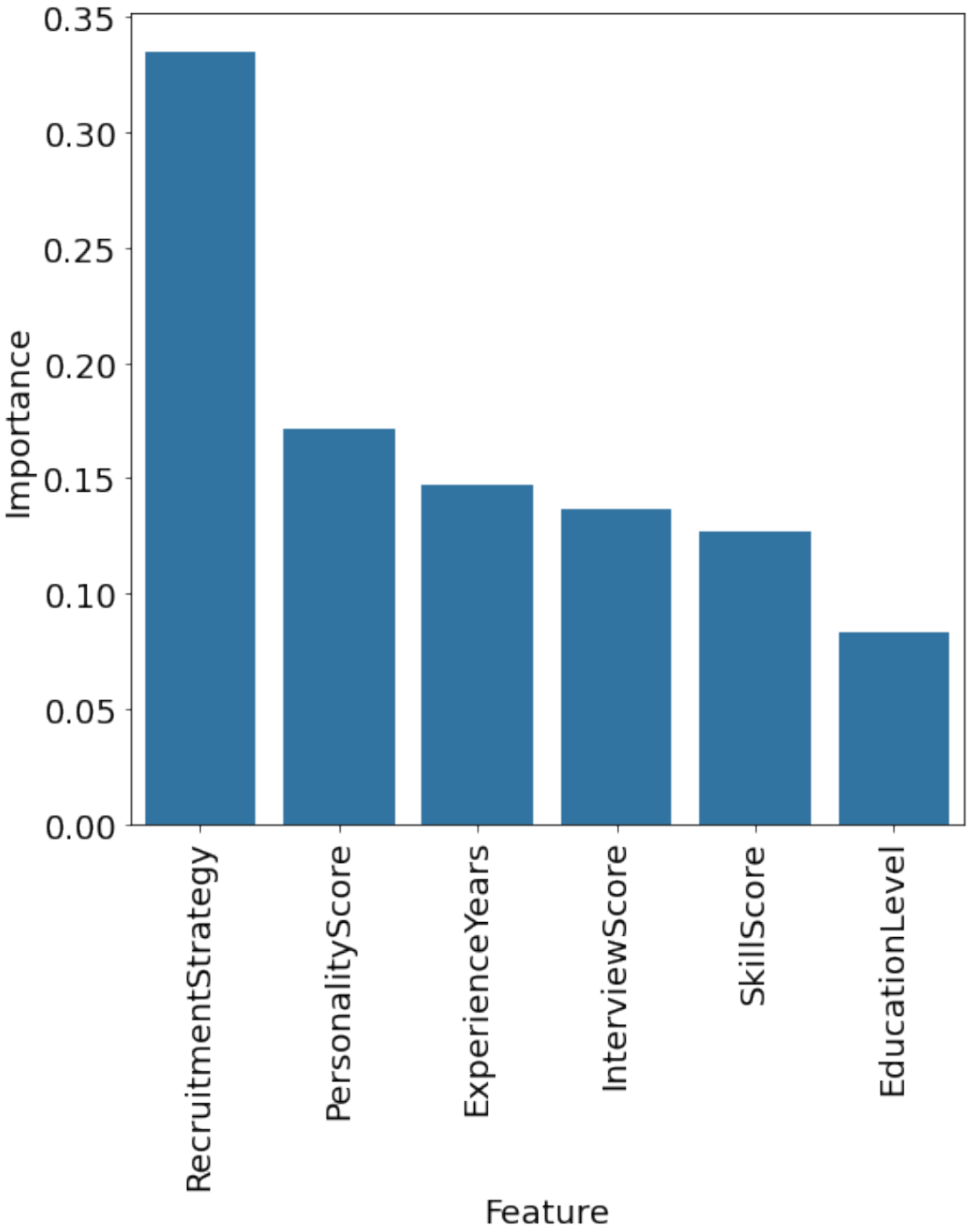
### 1.7 Interpreting the Decision Tree

Let's the feature importances in the decision tree model:

[160]:
```
dt_fit = DecisionTreeClassifier(random_state = 42,
                                criterion = criterion_opt,
                                max_depth = max_depth_opt,
                                max_features = max_features_opt).fit(X_train,␣
 ↪y_train)
importances_unsorted = dt_fit.feature_importances_
indices = np.argsort(-importances_unsorted)
importances = [importances_unsorted[i] for i in indices]
```

```
dt_features = [selected_features[i] for i in indices]
```

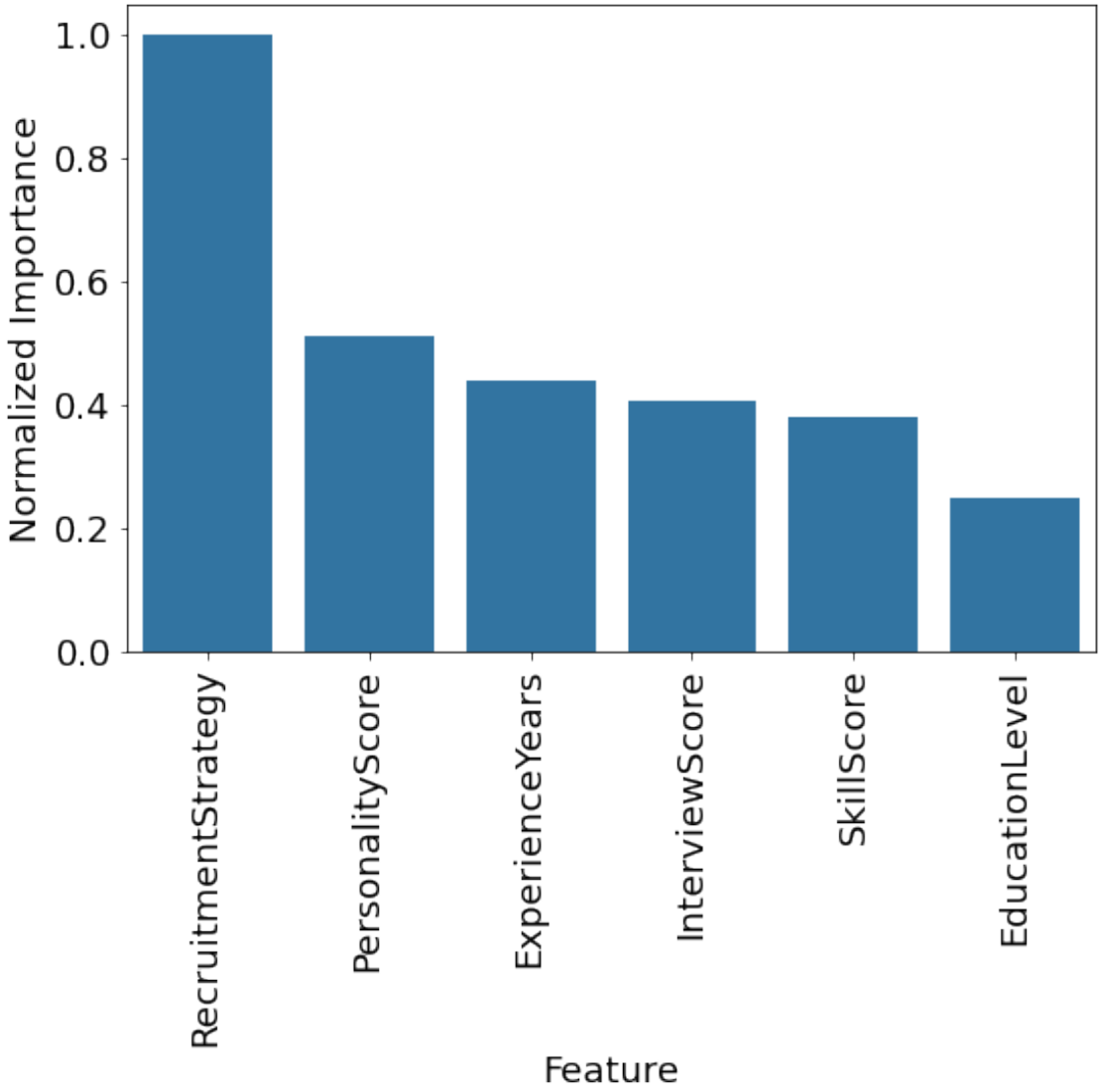
[154]:
```
importance_df = pd.DataFrame({"Feature": dt_features,
                              "Importance": importances,
                              "Normalized Importance": importances/
 ↪importances[0]})
fig, ax = plt.subplots(1, figsize = (8,8))
sns.barplot(data = importance_df, x = "Feature", y = "Importance", ax = ax)
plt.xticks(rotation = "vertical")
plt.show()
```




This plot becomes even more interpretable if we renormalize the feature importances such that the most important feature has unit importance:

[155]:
```
fig, ax = plt.subplots(1, figsize = (8,8))
```

```
sns.barplot(data = importance_df, x = "Feature", y = "Normalized Importance",␣
 ↪ax = ax)
plt.xticks(rotation = "vertical")
plt.tight_layout()
plt.savefig("./figures/normalized_importances.pdf")
plt.show()
```




We see that the most important feature in determining hiring outcome is `RecruitmentStrategy`. Combined with the distribution plots above, we find that candidates are most likely to be successful during aggressive hiring campaigns, while candidates are least likely to be successful during moderate hiring campaigns. The increased success of candidates during aggressive hiring campaigns could reflect, for example, a tendency of the hiring team to hire a candidate quickly with less regard for their qualifications, a commitment of the committee to fill the position, or an effort on the part

of the hiring committee to locate candidates who are more qualified and more likely to succeed.

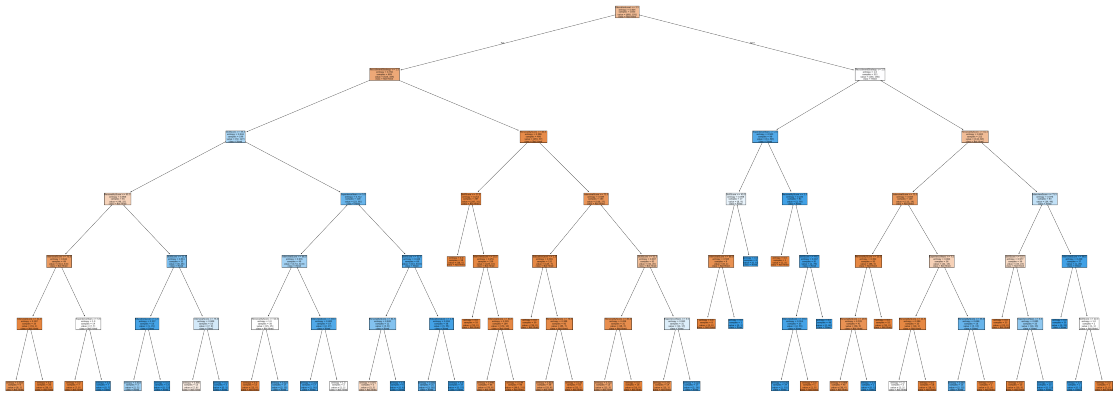
The next most important feature is `PersonalityScore`, which is only about half as important as `RecruitmentStrategy`. The relative feature importance decreases steadily to about 40% across `ExperienceYears`, `InterviewScore`, and `SkillScore`. It then drops more precipitously to about 25% for `EducationLevel`.

It is an interesting finding that the aggressiveness of the hiring strategy is the most important factor in determining candidate success. Future work would look at determining if this is because a candidate is always eventually hired during aggressive hiring campaigns, if hiring committees are committeed to filling positions quickly during aggressive campaigns and therefore are not as stringent about candidate qualifications, or if candidates are screened more effectively during the pre-interview vetting process and hence only more qualified candidates are considered during aggressive hiring campaigns. This question may be answered in part or entirely using the data available, but I will terminate the analysis here for the sake of time.

Future work might also look at the pathway to candidate success conditioned on a particular hiring strategy in order to determine if the factors leading to a candidate being hired are statistically distinct for people considered during aggressive and moderate hiring strategies.

Finally, let's take a look at the structure of the decision tree:

[69]:
```
fig, ax = plt.subplots(1,figsize = (50,20))
plot_tree(dt_fit, feature_names = selected_features, class_names = ["Not␣
 ↪Hired", "Hired"], filled = True)
plt.savefig("./figures/optimal_tree.pdf")
plt.show()
```



Though it is no more than six splits deep, it is still rather complex. It would be illuminating to examine the decision tree in more depth, but I will forgo this analysis for the sake of time.

[ ]:
```
```