Assignment #1

Jacqueline Heaton

Problem Formulation

The problem is to predict whether the matches will be successful or not. The input is all the information about the two people who are being matched (ideally, some people may have not given all the data), and the output will be the prediction of whether it is successful or not. This will require some preprocessing to replace missing values, and encoding to convert categorical features into numerical ones. Then it will require a model which predicts the likelihood of a successful match – in this case xgboost. The challenges will of course be that the parameters must be tuned to provide an optimal model that achieves a high accuracy in the prediction, and that the different search functions for finding parameters have differing strengths and weaknesses.

Documentation **(*This was before clarification. Some of this is relevant, but some may be extra. Skip to page 3 for using all 3 searches on just 1 type of model)**

Grid Search

Original score: 87%

- Changed n_est to [10, 100, 150], max_depth to [8,10,15,20]
    - Did best on n_est 100, may do better on higher so added 150, removed 20
    - Did best on max_depth 10, added 8 and 15 to see if does better with slightly lower or higher
    - Results: Slight improvement (87.7%), does best on n_est 150, max_depth 10
- Changed n_est to [100, 150, 200], max_depth to [8,10,15]
    - Still n_est=150, max_depth=10
- Changed n_est to [130,140,150]
    - Best is n_est=140, max_depth=10
    - Slight improvement (87.8%)
- Changed imputer strategy to mean
    - Accuracy slightly lower (87.6%), but low enough to attribute to noise/chance (less than 0.001%)
    - Best was with n_est=170, max_depth=10
    - Changed n_est to [150,170,200]
        - No change in results
    - Changed n_est = [165,170,175]
        - No change in results
        - Said best was n_est=165, max_depth=10
- Changed imputer strategy back to median, with n_est = [160,170,180]

    - No change in accuracy
    - Best was n_est=180, max_depth=15
- Set n_est = [170,180,190], max_depth=[10,15,20]
    - Slightly lower with n_est=190, max_depth=15, but small enough it could be due to noise (<0.002)
- Set n_est=[180,190,200]
    - No change, n_est=190, max_depth=15
- Change imputer strategy to most_frequent, n_est = [180,190,200]
    - Lower score by 1%, n_est=190, max_depth=20
    - Change max_depth to [15,20,25]
    - Same score, n_est=180, max_depth=20
        - Likely no difference in functionality between 180 and 190 estimators

Total time to run all parameters (Total expanse of n_est=[10, 100, 130, 135, 140, 150, 165, 170, 175, 180, 190, 200], Total expanse of max_depth=[8, 10, 15, 20]): 16.5 min

Random Search

Total expanse of n_est=[10, 100, 130,135,140,150,165,170,175,180, 190,200]

Total expanse of max_depth=[8,10,15,20]

Time: 2.3 min

- Imputer is mean
    - Best score is n_est=140, max_depth=8, accuracy slightly lower than best (87.6%)
    - Changed random seed, best score is n_est=200, max_depth=10, accuracy 87.7%
    - Added 210 to n_est, reran, best is n_est=165, max_depth=10, accuracy 87.8%
        - 210 was part of sample
    - Modified parameters to be n_est=[135,140,150,165,170,175,180,190,200,210], max_depth=[10,15], random seed back to 0
        - Accuracy of 87.8%
- Imputer is median, full expanse of parameters
    - Best score is n_est=130, max_depth=15, accuracy is 87.7%
    - Changed random seed, best score is n_est=140, max_depth=15, accuracy is 87.7%
- Imputer is most_frequent, full expanse of parameters
    - Best score is n_est=165, max_depth=20, accuracy is 87.6%
        - Scored 88.4% on public leaderboard – highest yet

SVM

Imputer is most_frequent. When predicting using binary, submission score of 62.5%. When using probability, 75.8%

Best score for probability was 82.5% for poly kernel, gamma = 2.3, degree = 8, C = 0.001

Time: 29.0 min

Note: it did not try rbf, linear was typically 3-5% lower than poly.

- Changed kernals to just poly and rbf
    - Best score was rbf at 82.8%, submission score of 75.8%
- Changed imputation strategy to mean
    - Best score still rbf at 82.8%, submission score of 82.0%

- Random search was much faster than the full Grid search, taking less than 3 minutes, while the Grid search took more than 16. The Random search provided results that were just as good if not better than the Grid search too, though the difference is small enough to be practically insignificant. The Random search does save a lot of time, allowing for the parameters to be tuned faster, however the Grid search does every combination, and you risk missing the best if you do Random. The best way would likely be to use Random search to narrow down the kinds of parameters that work well, and then use the Grid search to do a more intensive search for the best parameters once they've been narrowed down some.

Tuning using just 1 type of classifier

- All 3 imputer methods tried (mean, median, most_frequent)
- Used xgboost
- Grid:
    - N_estimators: [10, 100, 130, 135, 140, 150, 165, 170, 175, 180, 190, 200]
    - Max_depth: [8, 10, 15, 20]
    - Best score on training data was 87.8%
        - N_estimators = 165
        - Max_depth = 10
        - Imputer_strategy = mean
    - Score on public test data was 88.0%
- Random:
    - N_estimators: [10, 100, 130, 135, 140, 150, 165, 170, 175, 180, 190, 200]
    - Max_depth: [8, 10, 15, 20]
    - Best score on training data was 87.7%
        - N_estimators = 180
        - Max_depth = 15
        - Imputer_strategy = median
    - Score on public test data was 88.1%

- Bayesian:
    - N_estimators: [10, 100, 130, 135, 140, 150, 165, 170, 175, 180, 190, 200]
    - Max_depth: [8, 10, 15, 20]
    - Best score on training data was 87.7%
        - N_estimators = 145
        - Max_depth = 18
        - Imputer_strategy = median
    - Score on public test data was 87.6%

The Grid search took much longer than the other searches, performing a brute force search in trying all the combinations, and so took $O(n^2)$ time. It was able to cover all combinations of the input, and so was guaranteed to find the optimal solution. The Random search was much faster than the Grid search, and able to get a nearly identical score, with a slight improvement likely due to randomness. It took O(1) time, being dependent on n_iter, which was kept at its default value of 10. The Bayesian search was similarly very quick, running in O(1) time with n_iter set to its default of 50, however it attained a slightly lower score. While this lower score could be attributed to randomness, it is also possible that it converged to a local minimum, as due to the optimizing nature of the BayesianSearch it tends to concentrate on parameters that performed well, which may result in missing the best combination.

Looking at the parameters for the best score, mean and median both seem to result in high scores, as the brute force grid search used the mean for its best score, while both the random search and the Bayesian search used the median. The n_estimators are clearly best in the mid-high 100's range, and the optimal max_depth falls somewhere between 10 and 20.

Running 5 different models twice, using RandomSearch:

All models had the preprocessor parameter:

```
'preprocessor__num__imputer__strategy': ['mean','median']
```

The imputer strategy doesn't need to consider most_frequent since only mean and median produced the best results. Similarly, the n_estimators range was reduced to only include numbers near the optimal ones, which were grouped around the mid-high 100s. The max_depth remains as it was because its range for the optimal solutions covered most of the breadth of the selected parameters. The RandomSearch method was chosen over the BayesSearch because RandomSearch tended to produce better results, likely due to the BayesSearch getting stuck in a local optimum.

1st run:

XGBoost: Best score: 87.7%, parameters: N_est: 175, max_depth: 15, preprocessor: median

Leaderboard score: 88.2%

Full parameters:

```
'xgbClass__n_estimators': [140,150,165,170,175,180, 190,200],
'xgbClass__max_depth':[8,10,15,20]
```

Logistic Regression: Best score: 85.4%, parameters: solver: sag, penalty: l2, C: 1.5, preprocessor: median

Leaderboard score: 87.0%

Full parameters:

```
'lrClass__penalty': ['l1','l2','elasticnet'],
'lrClass__C':[0.1,0.5,1,1.5],
'lrClass__solver':['newton-cg', 'lbfgs', 'sag', 'saga']
```

Note:

- several parameters were incompatible with each other, however for the sake of testing, were kept for this 1st round of tuning
- for saga, and l1 or elasticnet, did not converge (c was irrelevant)
- lbfgs, and l1 or l2 did not converge (c was irrelevant)
- newton-cg, and l2 did not converge
- sag and l2 did not converge

Random Forest: Best Score: 84.7%, parameters: n_est: 100, max_depth: 10, preprocessor: mean

Leaderboard score: 85.2%

Full parameters:

```
'rfClass__n_estimators': [10,50,100],
'rfClass__max_depth':[8,10,15,20]
```

K-NearestNeighbors: Best Score: 81.3%, parameters: weights: distance, n_neighbors: 100, algorithm: ball_tree, preprocessor: mean

Leaderboard score: 82.6%

Full parameters:

```
'knnClass__n_neighbors': [10,50,100,150],
'knnClass__weights':['uniform', 'distance'],
'knnClass__algorithm':['auto', 'ball_tree', 'kd_tree', 'brute']
```

Note:

- Get ‘can’t use with sparse input’ warnings

MLP: Best score: 86.5%, parameters: solver: sdg, activation: tanh, hidden_layer_sizes: (100,150) preprocessor: median

Leaderboard score: 87.7%

Full parameters:

```
'mlpClass__activation':['identity', 'logistic', 'tanh', 'relu'],
'mlpClass__solver':['sgd', 'adam'] ,
'mplClass__hidden_layer_sizes':[(50,100),(100,100),(100,150)]
```

Note:

- convergence warning -> need to increase max_iter

2nd run:

XGBoost: Best score: 87.8%, parameters: n_estimators: 165, max_depth: 10, preprocessor: mean

Leaderboard score: 88.0%

Full parameters:

```
'xgbClass__n_estimators': [140,150,165,170,175,180],
'xgbClass__max_depth':[10,15,20]
```

In all runs, I only saw 180 estimators be used once for the best score, and never anything higher. The best runs tend towards the mid-100s, so removing some of the higher n_estimators values should allow for faster classification without impacting the score. Similarly, a max_depth of 8 was almost never used, and removing it shouldn’t impact the optimal score.

Logistic Regression: Best score: 85.6%, parameters: solver: sag, penalty: l2, c: 1.5, preprocessor: median

Leaderboard score: 86.9%

Full parameters:

```
'lrClass__penalty': ['l2'],
'lrClass__C':[0.1,0.5,1,1.5],
'lrClass__solver':['sag', 'saga']
```

The newton-cg and lbfgs showed poorer prediction accuracies, typically scoring in the low 80s, whereas the sad and saga solvers tended towards the mid-high 80s.Sag only supports l2 capabilities, which saga is compatible with l1, l2, and elasticnet. Since the scores were comparable with all 3 for saga, I've reduced the penalty to just l2. Additionally, I changed max_iter to 300 for better convergence.

Random Forest: Best Score: 85.0% parameters: n_estimators: 250, max_depth: 10, preprocessor: median

Leaderboard score: 85.8%

Full parameters:

```
'rfClass__n_estimators': [100,150,190,200,220,250,270],
'rfClass__max_depth':[8,10,15,20]
```

The classifier did poorly on the low n_estimators and the best score occurred at the maximum n_estimator value. I removed the numbers below 100 (10 and 50), and added some numbers that were higher, over a relatively wide range. This may affect what the ideal max depth is, so I left that parameter as is.

K-NearestNeighbors: Best Score: 82.5%, parameters: weights: uniform, algorithm: kd_tree, neighbors: 150, preprocessor: mean

Leaderboard score: 82.9%

Full parameters:

```
'knnClass__n_neighbors': [70,100,120,150],
'knnClass__weights':['uniform', 'distance'],
'knnClass__algorithm':['auto', 'ball_tree', 'kd_tree', 'brute']
```

Narrowed the range of n_neighbors to be closer to the one that scored the ideal, and removed the low performing n_neighbours values.

MLP: Best score: 85.6%, parameters: solver: adam, hidden_layer_sizes: (100,), activation: logistic, preprocessor: mean

Leaderboard score: 85.4%

Full parameters:

```
'mlpClass__activation':['logistic', 'tanh', 'relu'],
'mlpClass__solver':['sgd', 'adam'],
'mlpClass__hidden_layer_sizes':[(100,),(150,),(100,100),(100,150)]
```

Increased max_iter to allow for more convergence, as some activation functions did not converge. Removed identity activation which tended to score lower than the others, removed (50,100) because it tended to score slightly less, while the (100,100) and (100,150) were similar. Added (100,) and (150,) to see if number of layers being 1 is better

Questions

Why a simple linear regression model (without any activation function) is not good for classification task, compared to Perceptron/Logistic regression?

Data is often complex and not linearly separable. Simple linear regression is only capable of separating 2 dimensional data, and uses only a line (linear function) to try and predict. This doesn't work for complex classification tasks that involve many dimensions/features.

What's a decision tree and how it is different to a logistic regression model?

A decision tree splits the data into subsets based on the values of various features. This reduces the full data set into many smaller dataset, with the goal of reducing the data until all data points that fall within one of the leaf/terminal nodes have the same answer. Logistic regression predicts the output based on a function, and works well with continuous data (infinite possible answers)

We discussed three variants of decision tree in our lecture, what are their differences?

The variants were the ID3, C4.5, and CART algorithms.

ID3 uses the information gain algorithm to decide which features to use to split the data first. No pruning is done, and it treats data as categorical, deciding based on the number of different values that feature can take and the way the answers are separated among them.

C4.5 uses the gain ratio method which is a version of information gain with reduced bias towards high-branch attributes. This typically results in a more even distribution, however it may overcompensate due to its choosing method. This method deals better with noisy and unclean data, and can use both categorical and numeric attributes.

CART uses the gini index to try to assess the likelihood of an attribute to be 'impure'. If an attribute has a wide range of answers compared to other attributes, then it is more likely to be considered impure and irrelevant to the classification. This is a more robust type of decision tree that can use both categorical and numeric attributes, and can deal with noisy and unclean data.

What is the difference between the random forest model, and a bagging ensemble of CART models?

The random forest uses many decision trees, each with a different subset of attributes. Both use a subset of the total dataset, but bagging uses all the attributes.

What is the difference between bagging and boosting?

In bagging the samples and predictors are all equal, while boosting uses previous knowledge and adjustable weights to give classifiers a voting ability proportional to their accuracy. This allows later classifiers/predictors to specialize in more difficult objects, but can result in overfitting.

How can we use linear models to solve non-linear problem? (two approaches)

Some data is not linearly separable, but can be transformed into another space where is can be separated. For example, if we have 2 sets of data with 2 coordinates, one in the shape of a circle, and the other in a larger, hollow circle, this can be transformed to separate the 2 circles by giving the data points the distance from the center as a 3rd feature. This now 3 dimensional data can be separated with a linear model.

The second method uses a multi-layer perceptron, which uses multiple layers of linear activation functions to classify non-linear data. This can help separate data that has multiple attributes making it belong to the same group (ex: if either purple and green make a flower part of group one, and either red or black make a flower part of group two), that can't be isolated using 1 line, but may be isolated using several.

Code:

```
# -*- coding: utf-8 -*-
"""A1_walk_through.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/173dGt3z3oS4w4VK1z0a3PQmxICg32EAT
"""

import pandas as pd
import numpy as np
from sklearn.metrics import f1_score
from pprint import pprint

from sklearn.compose import ColumnTransformer
from sklearn.datasets import fetch_openml
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV,
RandomizedSearchCV
from xgboost.sklearn import XGBClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from skopt.space import Real, Categorical, Integer
from sklearn.linear_model import LogisticRegression
```

```
data = pd.read_csv('train.csv')
data_test = pd.read_csv('test.csv')

# prints out the dimenstions of the train data
data.shape

# detects missing values, sums, plots histogram
data.isnull().sum().hist()

# just the histogram for this column
data['match'].hist()

# if you haven't installed xgboost on your system, uncomment the line below
# !pip install xgboost
# if you haven't installed bayesian-optimization on your system, uncomment the line
below
#!pip install scikit-optimize

# remove column match (this is the answer column)
x = data.drop('match', axis=1)
# separate numeric and categorical features
features_numeric = list(x.select_dtypes(include=['float64']))
features_categorical = list(x.select_dtypes(include=['object']))
# store answers separately
y = data['match']

print(features_categorical)


np.random.seed(1)

# make numerical imputer as pipeline - more streamlined
# strategy will be overwritten
transformer_numeric = Pipeline(
    steps=[
        ('imputer', SimpleImputer(strategy='mean')),
        ('scaler', StandardScaler())]
)

# make categorical imputer, replace with 'missing'
# encode to convert values to numbers
transformer_categorical = Pipeline(
    steps=[
        ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
        ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ]
)

# combines pipelines and apply to data
preprocessor = ColumnTransformer(
    transformers=[
        ('num', transformer_numeric, features_numeric),
        ('cat', transformer_categorical, features_categorical)
    ]
)
```

```
# does preprocessing and classification (gradient boosted decision trees)
# initializes XGBClassifier
pipeline_XGB = Pipeline(
    steps=[
        ('preprocessor', preprocessor),
        ('xgbClass', XGBClassifier(
            objective='binary:logistic', seed=1))
    ]
)

# uses same preprocessor, initializes logistic regression classifier
# sets iterations to 600 with tol to 0.005 (represents when it considers it
converged)
pipeline_LR = Pipeline(
    steps=[
        ('preprocessor', preprocessor),
        ('lrClass', LogisticRegression(max_iter=600, tol=0.005))
         ]
)

# initializes random forest classifier
pipeline_RF = Pipeline(
    steps=[
        ('preprocessor', preprocessor),
        ('rfClass', RandomForestClassifier())
         ]
)

# initializes K nearest neighbors classifier
# - chosen instead of svm due to familiarity with svm
pipeline_KNN = Pipeline(
    steps=[
        ('preprocessor', preprocessor),
        ('knnClass', KNeighborsClassifier())
         ]
)

# initializes MLP with set max iterations and learning rate initialization
# due to convergence issues, initial learning rate set to higher value
pipeline_MLP = Pipeline(
    steps=[
        ('preprocessor', preprocessor),
        ('mlpClass', MLPClassifier(max_iter=600, learning_rate_init=0.01))
         ]
)

# dictionary of parameters

# `__` denotes attribute
# (e.g. my_classifier__n_estimators means the `n_estimators` param for
`my_classifier`
#  which is our xgb)
param_XGB = {
    'preprocessor__num__imputer__strategy': ['mean','median'],
```

```
    'xgbClass__n_estimators': [140,150,165,170,175,180],
    'xgbClass__max_depth':[10,15,20]
}

# the parameters chosen for logistic regression
param_LR = {
    'preprocessor__num__imputer__strategy': ['mean','median'],
    'lrClass__penalty': ['l2'],
    'lrClass__C':[0.1,0.5,1,1.5],
    'lrClass__solver':['sag', 'saga']
}

# the parameters chosen for random forest
param_RF = {
    'preprocessor__num__imputer__strategy': ['mean','median'],
    'rfClass__n_estimators': [100,150,190,200,220,250,270],
    'rfClass__max_depth':[8,10,15,20]
}

# parameters for k nearest neighbors
param_KNN = {
    'preprocessor__num__imputer__strategy': ['mean','median'],
    'knnClass__n_neighbors': [70,100,120,150],
    'knnClass__weights':['uniform', 'distance'],
    'knnClass__algorithm':['auto', 'ball_tree', 'kd_tree', 'brute']
}

# parameters for multi layer perceptron
param_MLP = {
    'preprocessor__num__imputer__strategy': ['mean','median'],
    'mlpClass__activation':['logistic', 'tanh', 'relu'],
    'mlpClass__solver':['sgd', 'adam'],
    'mlpClass__hidden_layer_sizes':[(100,),(150,),(100,100),(100,150)]
}

"""
# exhaustive search over parameters, uses the pipeline made above
grid_search = GridSearchCV(
    full_pipline, param_grid, cv=5, verbose=3, n_jobs=2,
    scoring='roc_auc') # computes area under curve from prediction scores

grid_search.fit(x, y)

# print out best average score found in search and the parameters for it

print('best score {}'.format(grid_search.best_score_))
print('best score {}'.format(grid_search.best_params_))

# prepare submission:
submission = pd.DataFrame()
submission['id'] = data_test['id']
submission['match'] = grid_search.predict_proba(data_test)[:,1]
submission.to_csv('RandomSearch.csv', index=False)
"""
```

```
# make array of pipelines and parameters to iterate through
pipelines = [pipeline_XGB, pipeline_LR, pipeline_RF, pipeline_KNN, pipeline_MLP]
params = [param_XGB, param_LR, param_RF, param_KNN, param_MLP]
names = ['XGB', 'LR', 'RF', 'KNN', 'MLP']

# iterate through pipelines and parameters
# use random search - produces better results than Bayesian, likely due to getting
stuck in local optimums
for i in range(5):
    # uses same parameters and model
    random_search = RandomizedSearchCV(
        pipelines[i], params[i], cv=5, verbose=3, n_jobs=2,
        scoring='roc_auc')

    random_search.fit(x, y)
    print('best score {}'.format(random_search.best_score_))
    print('best score {}'.format(random_search.best_params_))

    # prepare submission:
    submission = pd.DataFrame()
    submission['id'] = data_test['id']
    submission['match'] = random_search.predict_proba(data_test)[:,1]
    submission.to_csv('RandomSearch' + names[i] + '.csv', index=False)


"""
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
from sklearn.svm import SVC

# bayesian search parameters for xgboost
# must be in ranged form
param_bayes = {
    'preprocessor__num__imputer__strategy':
Categorical(['most_frequent','mean','median']),
    'my_classifier__n_estimators': Integer(10, 210, 'log-uniform'),
    'my_classifier__max_depth':Integer(5, 30, 'log-uniform')
}

# chooses values based on optimized approach
# essentially random but skewed towards high performing ones
bayes_search = BayesSearchCV(
    full_pipline, param_bayes, cv=5, verbose=3, n_jobs=2,
    scoring='roc_auc',
    n_iter=3,
    random_state=0,
    refit=True,

)

bayes_search.fit(x, y)

# print out best score and its parameters
print('best score {}'.format(bayes_search.best_score_))
print('best score {}'.format(bayes_search.best_params_))
```

```
"""

"""
# make another pipeline, same preprocessor but using an SVM instead of decision trees
SVC_pipline = Pipeline(
    steps=[
        ('preprocessor', preprocessor),
        ('my_svc', SVC(class_weight='balanced', probability=True))
    ]
)
# SVC has a class_weight attribute for unbalanced data


# samples a fixed number of parameters (not exhaustive like GridSearchCV) from
distribution
# Sets svm parameters: C-regularization, kernal-type of kernal, degree-for polynomial
kernal, gamma-influence
# define ranges for bayes search
bayes_search = BayesSearchCV(
    SVC_pipline,
    {
        'my_svc__C': Real(1e-6, 1e+6, prior='log-uniform'),
        'my_svc__gamma': Real(1e-6, 1e+1, prior='log-uniform'),
        'my_svc__degree': Integer(1,8),
        'my_svc__kernel': Categorical(['poly', 'rbf']),
    },
    n_iter=3,
    random_state=0,
    verbose=3,
    refit=True,
)

bayes_search.fit(x, y)

# print out best score and its parameters
print('best score {}'.format(bayes_search.best_score_))
print('best score {}'.format(bayes_search.best_params_))

# lists which parameters were chosen
print('all the cv scores')
# pretty printing
pprint(bayes_search.cv_results_)



# prepare submission:
submission = pd.DataFrame()
submission['id'] = data_test['id']
submission['match'] = bayes_search.predict_proba(data_test)[:,1]
submission.to_csv('BayesSearch.csv', index=False)

"""
```