BACT HACKATHON

Team Cucumber:

Lenard Oh, Chua Kai Jun, Sean Hardjanto

3 Aug 2024

Table of Contents

- Data Imbalance and Techniques
- Experimentation with SMOTE and UnderSampling
- Final Model Decision

The Model

- Model Selection Process
- Hyperparameter Tuning with GridSearchCV
- Final Model Performance

Appendices

- **Appendix A: Features**
    - Figure 1: Distribution of Job Salary and Working Years
    - Figure 2: Distribution of Engineered Salary Difference
    - Figure 3: Distribution of Engineered Skill Match
- **Appendix B: Synthetic Generation**
    - Figure 1: SMOTE and UnderSampling Parameters Graph
- **Appendix C: Model Metrics**
    - Comparison of Various Models
- **Appendix D: Code Snippets**
    - Final Code for Preprocessing and Model Training
    - Code for Hyperparameter Tuning
    - Best Parameters

Approach

Our aim was to examine three of the potential approaches given: feature selection and engineering, synthetic generation of minority class data, and ML model training. We used the train_main.csv data to train our model. We also included various external libraries such as(the pandas and scikit-learn libraries in Python3) to build our model, aid in data visualisation and model accuracy.

Methodology

Data processing

Before developing the model, we needed to identify relevant variables and transform ambiguous features into actionable data. Since we decided not to use LLMs, we began by eliminating variables that would require LLM processing. Next, we engineered features from existing ones, such as the number of skills listed by each candidate and matches between candidate skills and job requirements.

Synthetic Generation

We also implemented synthetic generation to improve the imbalance of the data. Initially, the minority class of ‘result=0’ made up only about 15% of the test set compared to the majority class of ‘result=1’ at 85%. To counteract the inconsistencies arising from this imbalanced data, we attempted to use SMOTE to generate more minority data, thus oversampling the minority data and undersampling the majority data to balance the data set[1]. Interestingly, testing different values of SMOTE and UnderSampling parameters showed that the model trained on undoctored data was most accurate (Appendix B). Hence our final model does not use either technique to attempt to balance the training data.

Model Evaluation

Lastly, we selected the RandomForestClassifier model after comparing the accuracy of several commonly-used binary classification models. We then used the hyperparameter tuning tool GridSearchCV to find the most optimal parameters. We used a train-test split of 80%-20% to evaluate our model - 80% of data was used for training while the remaining 20% of the data was set aside for model evaluation.

We used the following metrics to measure model performance:

---

[1] Jason Brownlee, “Smote for Imbalanced Classification with Python,” MachineLearningMastery.com, March 16, 2021, https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/.

- ➢ Accuracy - proportion of test set we predicted correctly
- ➢ Precision - TruePositive/(TruePositive+FalsePositive)
- ➢ Recall- TruePositive/(TruePositive+FalseNegative)
- ➢ F1 score - (2*Precision*Recall)/(Precision + Recall)

Since the data is highly imbalanced, we considered F1 score to ensure that the model is not skewed towards either recall or precision[2].

Feature Selection/Engineering

All the code for feature selection and engineering can be found in our 'preprocessing' function. This function was used to pre-process both the test_main and train_main data, to prepare them for model input.

Our final list of features is: ['skill_match', 'working_years', 'avg_salary', 'salary_diff', 'skill_count', 'job_skill_count', 'skill_over_job']

Numerical Features

The working_years, job_min- and job_max-salary for each job could be indicative of competitiveness, where high experience or salaried jobs may have lower success rates. As shown in **Appendix A: Fig. 1**, among unsuccessful applicants there were more jobs with high min/max salaries and high working years. We also engineered an 'avg_salary' column, simply $\frac{max\ job\ salary + min\ job\ salary}{2}$. When training the model, this 'avg_salary' feature produced higher accuracy and f1-score than using just 'max_job_salary' and 'min_job_salary' as separate features.

The column 'expected_salary' was missing over 40% of entries. We chose to engineer a feature called 'salary_diff', calculated as $(\frac{avg\ salary - expected\ salary}{avg\ salary}).fillna(0)$. People whose expected salaries were higher than the job's average had negative values, while those who expected lower than average had positive values. Missing values were filled as 0. Although there was no pronounced difference in salary_diff between result=0 and result=1, shown in **Appendix A: Fig. 2**, keeping this feature still improved model accuracy.

All numerical features were scaled using StandardScaler[3] and clipped at -5 and 5 to remove highly anomalous data.

---

[2] "F1_score," scikit, accessed August 3, 2024, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html.
[3] "Standardscaler," scikit, accessed August 3, 2024, https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

Categorical Features

There was a huge variety of skills listed in ‘skill_name’, ‘cv_skill_name’, and ‘job_skill_name’. As such, even after grouping similar skills, one-hot encoding each possible skill for each candidate would add far too many columns. We decided to count the number of skills overlapped between the candidate’s skills and the skills required by the job itself. This ‘skill_match’ feature, shown in **Appendix A: Fig. 3**, was found to improve model accuracy. Additional features of ‘skill_count’, the quantity of skills each candidate listed; ‘job_skill_count’, the quantity of skills required by the job; and ‘skill_over_job’, simply $\frac{skill\ count}{job\ skill\ count}$, made small improvements to the model’s accuracy.

We also attempted to engineer features from ‘function_name’ and ‘job_name’. Due to the high cardinality, which makes one-hot encoding unviable, we attempted to target-encode these columns using the proportion of successes for each job name (e.g. 15% of Sustainability Senior Managers were successful in the train set, so we replaced each entry with 0.15). However, this assumes that the train_main data is an accurate representation of the field as a whole, and that the test_main data has the same list of job names. Additionally, these features considerably reduced model accuracy, so they were discarded.

Synthetic Generation

The train_main data was made up of about 85% of result=0 data and 15% of result=1 data. We aimed to train the model on a more balanced dataset for better performance on the minority class.

We experimented with the Synthetic Minority Oversampling Technique (SMOTE) to generate more of the result=1 data, and Random Undersampling to randomly reduce the amount of result=0 entries when training the model. For fairness, we set aside 20% of the data as a test set *before* performing the synthetic generation techniques. As shown in **Appendix B**, when testing different combinations of SMOTE parameter (the proportion of result=1 to be created) and undersampling parameter (the percentage of result=0 to use), we found that not using either technique produced the best accuracy, precision and f1 scores. We thus did not use synthetic generation in our final model.

The Model

When researching for models to use, we came across 4 possible models: Logistic Regression, Nonlinear SVM, Random Forest, and Neural Network[4].

Model Selection

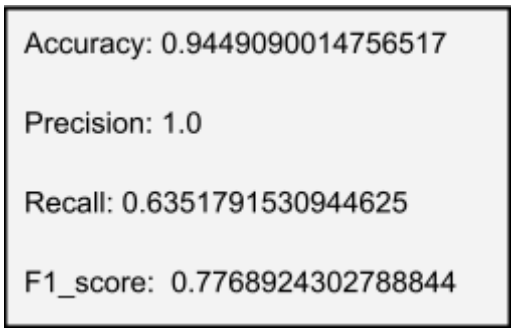
As shown in **Appendix C**, trying various commonly-used binary classification models showed that RandomForestClassifier from the Scikit-learn library produced the best metrics.

Hyperparameter Tuning

Hyperparameters play a pivotal role in determining a model's structure, functionality, and performance. To achieve this, as seen in **Appendix D: Fig. 2D**, we used GridSearchCV to tune the hyperparameters to optimise the model's performance to achieve the best possible results (**Appendix D: Fig. 2E**).

Final Model Performance

After all the measures we took in order to counteract the imperfection of the given data and other challenges, our final model's performance resulted in an increased accuracy of 0.9449090014756517.



[4] Andrii Gozhulovskyi, "Choosing a Model for Binary Classification Problem," Medium, September 1, 2022, https://medium.com/@andrii.gozhulovskyi/choosing-a-model-for-binary-classification-problem-f211f7a4e263.

Appendix A: Features

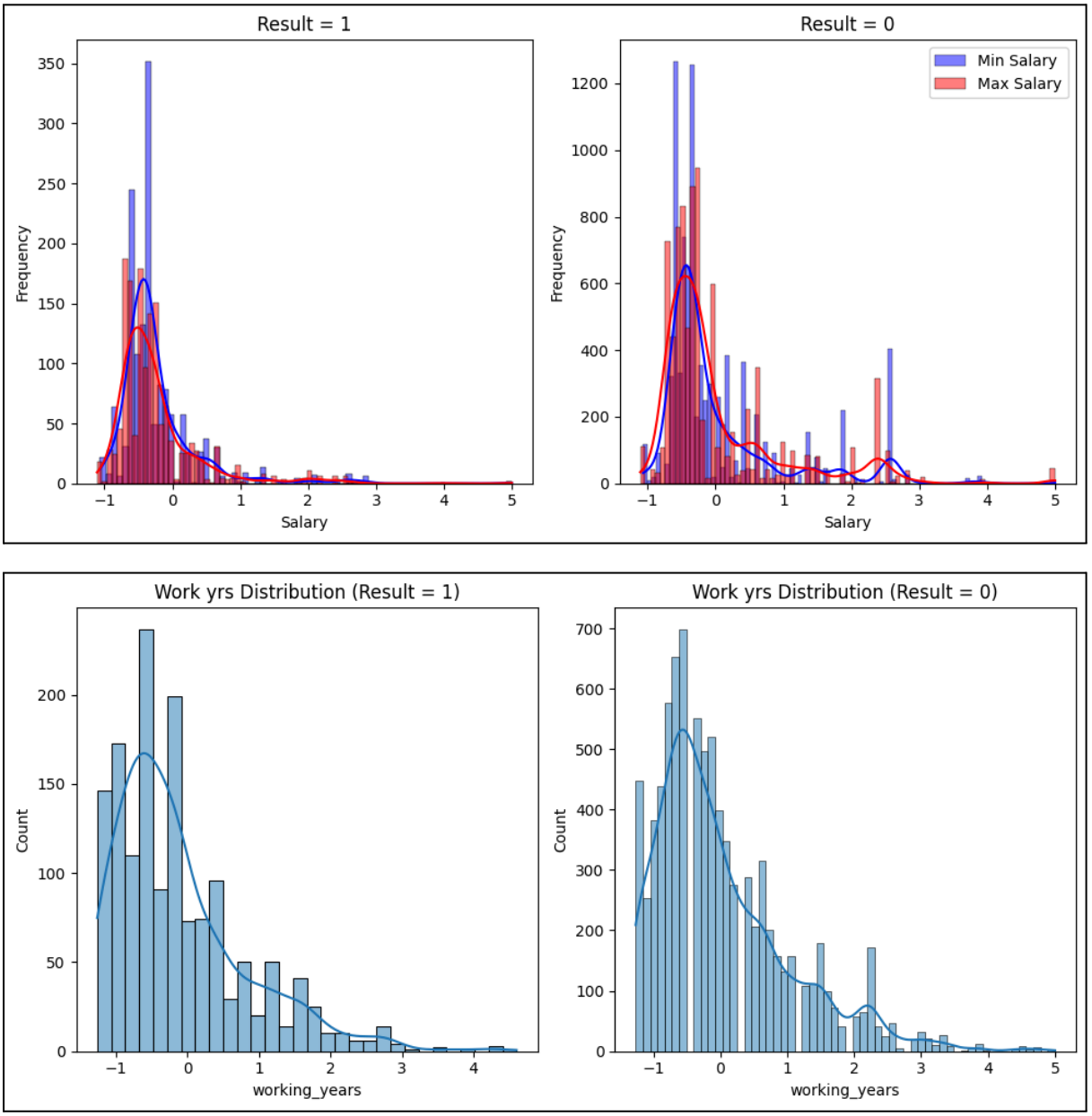


Figure 1: A comparison of the distribution of job_salary and working_years between successful and unsuccessful applicants

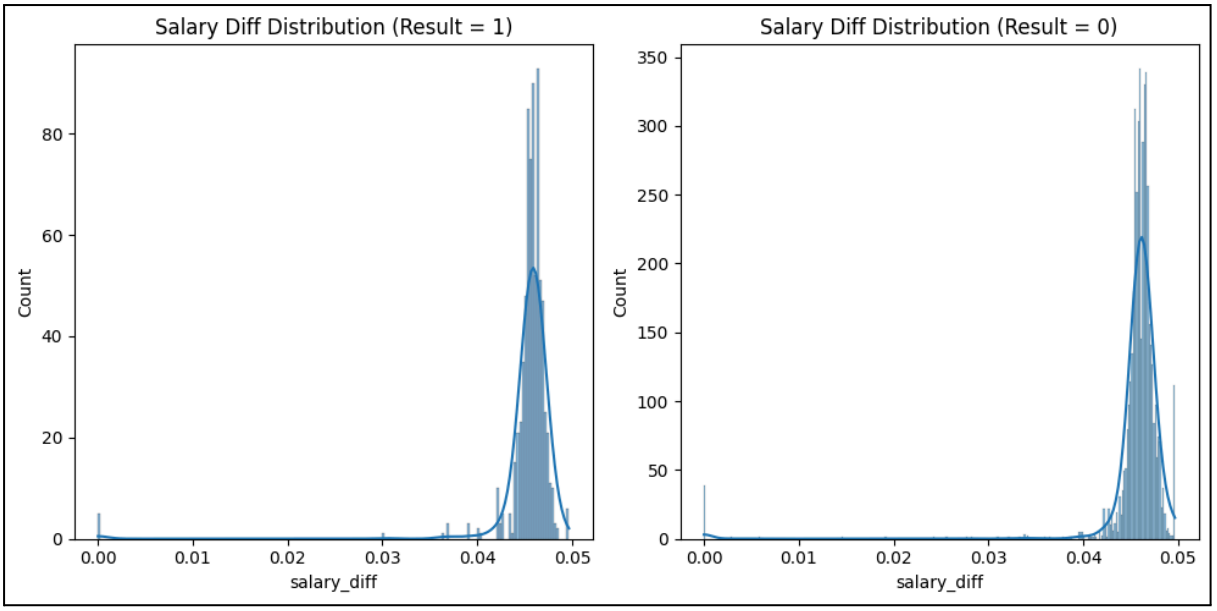


Fig 2: A comparison of the distribution of the engineered salary_diff feature

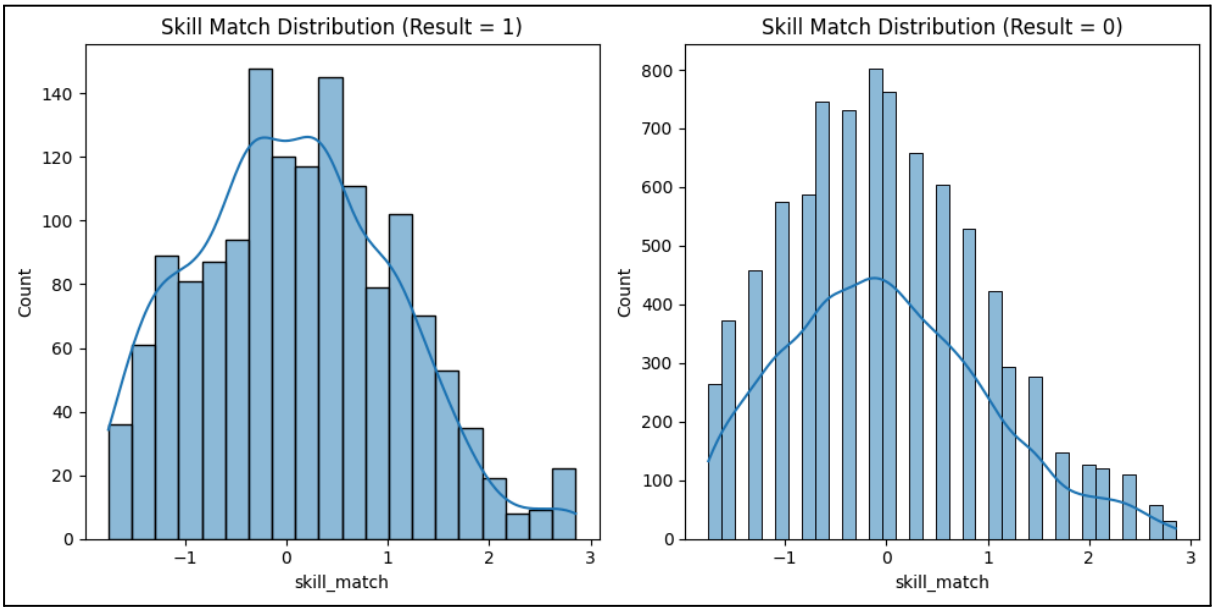


Fig 3: A comparison of the distribution of the engineered skill_match feature

Appendix B: Synthetic Generation

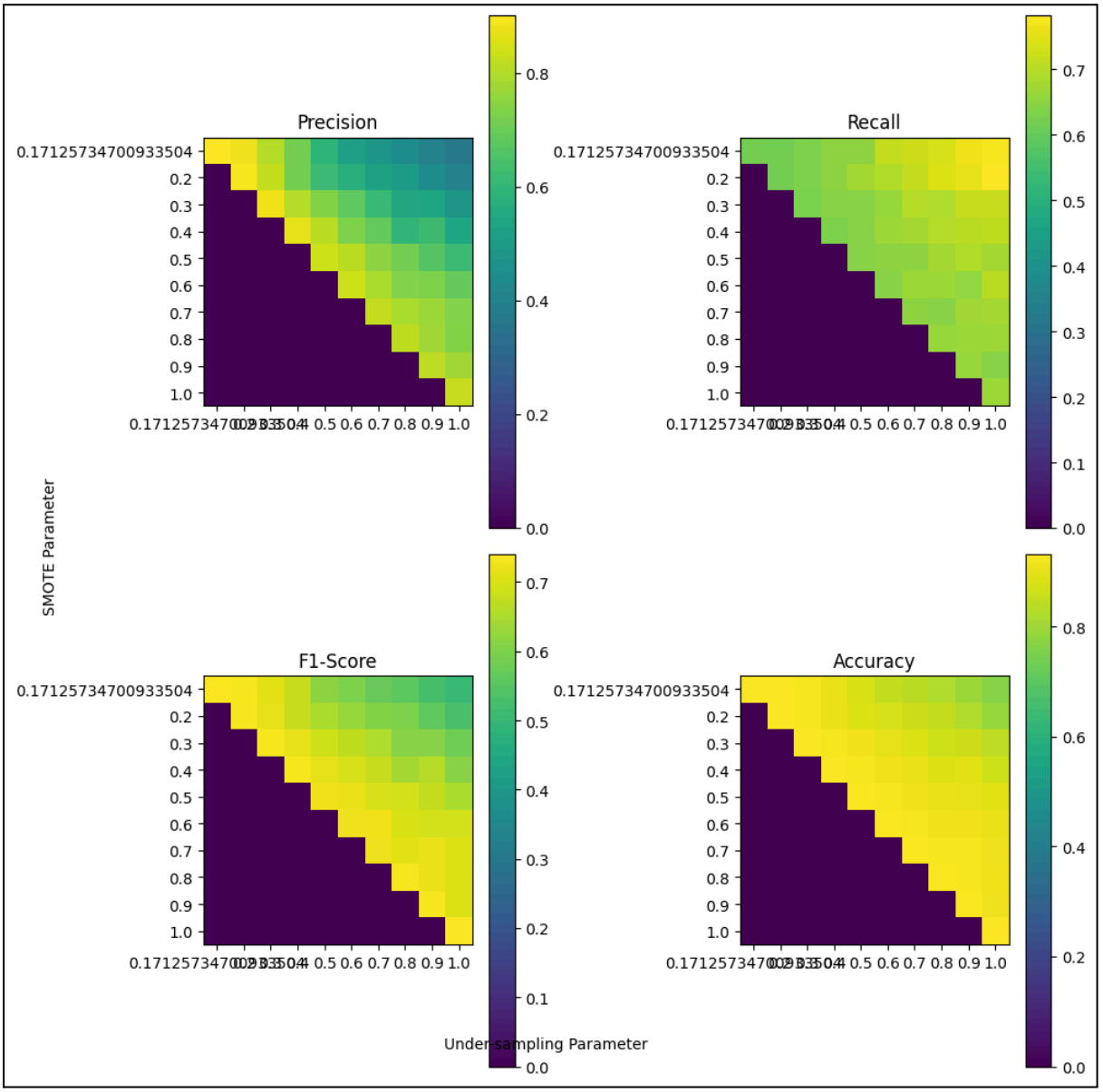


Fig 1: Graph of SMOTE Parameters and Undersampling Parameters.

X-axis: UnderSampling parameter, Y-axis: SMOTE Parameter.

The entry 0.171257… represents the control, where neither SMOTE or UnderSampling was used. The graph shows that (0.171,0.171) had the best accuracy, F1, and precision, suggesting that we should not use undersampling or SMOTE at all.

Appendix C: Model Metrics

	RandomForest	**Logistic Regression**	**Non-Linear SVM**	**Neural Network (tf.keras)**
Accuracy	0.81317	0.72016	0.72181	0.74518
Precision	0.76444	0.55172	0.61765	0.80935
Recall	0.49711	0.09249	0.06069	0.23106
F1	0.60245	0.15841	0.11053	0.35949

Appendix D: Code Snippets

Final Code

```
import numpy as np

import pandas as pd

from sklearn.preprocessing import StandardScaler

import ast

from sklearn.ensemble import RandomForestClassifier



#Pre-processing function

def preprocessing(df):

  df['skill_name'] = df['skill_name'].apply(ast.literal_eval)

  df['job_skill'] = df['job_skill'].apply(ast.literal_eval)

  df['cv_skill_name'] =
df['cv_skill_name'].apply(ast.literal_eval)



  skill_match_count = []

  skill_count = []

  job_skill_count = []

  skill_over_job = []

  for i in range(len(df['skill_name'])):

    candidate_skills = set(df['skill_name'][i] +
df['cv_skill_name'][i])

    job_reqs = set(df['job_skill'][i])

    skill_count.append(len(candidate_skills))

    job_skill_count.append(len(job_reqs))

    skill_over_job.append(len(candidate_skills)/len(job_reqs))
```

```
skill_match_count.append(len(candidate_skills.intersection(job_req
s)))
  df['skill_match'] = skill_match_count
  df['skill_count'] = skill_count
  df['job_skill_count'] = job_skill_count
  df['skill_over_job'] = skill_over_job

  df['avg_salary'] = (df['job_max_salary'] + df['job_min_salary'])
/ 2
  df['salary_diff'] =  ((df['avg_salary'] -
df['expected_salary']).fillna(0))/df['avg_salary']
  df['expected_filled'] = df['expected_salary'].notna()

  features = ['skill_match', 'working_years',
              'avg_salary', 'salary_diff',
              'skill_count', 'job_skill_count', 'skill_over_job'
              ]
  X = df[features]
  scaler = StandardScaler()
  columns = X.columns
  X_norm = scaler.fit_transform(X)
  X = pd.DataFrame(X_norm, columns = columns)
  X = np.clip(X, -5, 5)
  X['salary_diff'] = np.clip(X['salary_diff'], 0, 5)
  return X
```

```
#Final Model

train_main = pd.read_csv("/content/train_data_main.csv", engine =
'python')

test_main = pd.read_csv("/content/test_data_main.csv")


best_params = {

    'bootstrap': True,

    'max_depth': None,

    'max_features': 'sqrt',

    'min_samples_leaf': 1,

    'min_samples_split': 2,

    'n_estimators': 330

    }

final_model = RandomForestClassifier(**best_params,
random_state=42)

y_train_main = train_main['result']

X_train_main = preprocessing(train_main)

X_test_main = preprocessing(test_main)

final_model.fit(X_train_main, y_train_main)

y_pred = final_model.predict(X_test_main)


submission = pd.DataFrame(data={' ':test_main['Unnamed: 0'],
'result' : y_pred})

submission.to_csv('submission.csv', index=False)
```

Fig 1D: Final code

Code for Hyperparameter Tuning

```
from sklearn.model_selection import GridSearchCV

# Create the parameter grid based on the results of random search

param_grid = {

    'bootstrap': [True, False],

    'max_depth': [None, 2, 4, 6, 8, 10],

    'max_features': ['sqrt','log2',None],

    'min_samples_leaf': [1,2,3,4],

    'min_samples_split': [2,4,6,8],

    'n_estimators': [30*i for i in range(3,12)]

}

# Create a based model

rf = RandomForestClassifier()

# Instantiate the grid search model

grid_search = GridSearchCV(estimator = rf, param_grid = param_grid,

                           cv = 3, n_jobs = -1, verbose = 2)

grid_search.fit(train_X, train_y)
```

Fig 2D: Tuning Model Hyperparameters

```
best_params = {'bootstrap': True,

 'max_depth': None,

 'max_features': 'sqrt',

 'min_samples_leaf': 1,

 'min_samples_split': 2,

 'n_estimators': 330}
```

Fig 2E: Best Parameters