Introduction to Machine Learning Semester B

Lecturer: Dor Bank

Teaching Assistant: Ron Elias

Due Date: 12.8.2024

Project Summary

Natanel Druyan 212356661

Naama Tanchuma 322432071

Assumptions:

- Prev_salary is measured in Dollars
- Salary increases with years of experience
- Salary is yearly
- Individuals finish High School at age 18
- Individuals with work experience are over 18
- As of right now, we use seed 4
- All the applicants have worked in the past, not even necessarily in High-Tech. This assumption was made because the lowest prev_salary is 1 and not 0, hence there was probably a mistake when adding the data.

<u>Summary Report</u>

Assumptions that were made:

- Prev_salary is measured in Dollars: this is to ensure that we are working in uniform measurements, as it would be difficult to draw conclusions from prev_salary if all of the salaries were measured in local currency.
- Salary increases with years of experience: this is true for most occupations.
- Salary is yearly: this is a fair assumption as it is the most common way of measuring salaries.
- Individuals finish High School at age 18: this is standard globally.
- Individuals with work experience are over 18: while this is not necessarily the case 100% of the time, most HighTech positions require a degree which is incredibly rare for someone 18 and under to have.
- In order for reproducibility, we set all random_state parameters to seed 4
- All the applicants have worked in the past, not even necessarily in High-Tech. This assumption was made because the lowest prev_salary is 1 and not 0, hence there was probably a mistake when adding the data.

In this summary, we will recount the process of creating the project in its entirety, including the various challenges we faced and the steps taken to overcome them.

We started by analyzing the raw data as we received it, this included going over the features and familiarizing ourselves with them. The best way to achieve this would be through visualization, and so we made a correlation matrix of the data, as well as histograms of every individual feature. It was then decided that we were unable to draw any meaningful conclusions by looking at the data as it was, so we moved on to creating the models themselves. The models we decided to implement were: K-nearest neighbor, Logistic Regression, Multi-Layer Perceptron, and Random Forest.

Before implementing the models however, we were first required to fill in any missing variables in the data. This is crucial as it would severely hinder the effectiveness of the models otherwise, and some won't work at all. There were a great deal of missing variables in the data. For each feature we chose a different method in order to fill in the missing values as accurately as possible. Provided is a summary of the various methods used to deal with the NaN values within the data:
For "worked_in_the_past" we filled all the NaN values with "F" for False.
For "age_group" we took the number 18 as a base and then added that individual's "years_of_experience". If the sum was under 35 then the NaN value would be replaced with "young", otherwise it would be replaced with "old".
For "Disability" we calculated the probability of an individual having a "yes" in the "Disability" column using the ratio of "yes"s to "no"s in the data. Then we filled in the NaNs in the column in accordance with the calculated probability by generating random numbers between 0 and 1, we classified them by checking if they were higher than the calculated probability or not. If they were higher, that meant that the individual was't disabled, otherwise we counted the individual as disabled.
For "is_dev" we used a function to replace the NaN values with either "developer" or "non-developer" in accordance with their appearance ratio, similarly to how we did it with "Disability".
For "stack_experience" we used the IterativeImputer function in sk.learn. IterativeImputer fills missing values by modeling each feature with missing values as a function of other features in a round-robin fashion.
For "education" all NaN values were replaced with "other".
For "sex" like how we dealt with the missing values in "disability", we calculated a ratio between the three possible options: "Male", "Female", and "Other". The NaN values would then be replaced by these options in accordance with their calculated likelihood.

For “mental_issues” we first calculated the ratio of “yes”s to “no”s in the column data. After which we used a function to randomly replace the NaN values in the column with either “yes” or “no” in accordance with the calculated probability, similarly to the method we used for the “Disability” feature.
For “years_of_experience” we replaced all NaN values with the calculated median of the existing data in “years_of_experience”.
For “prev_salary” we used the IterativeImputer function in sk.learn.
For “A” and “B”, which are unknown features, we replaced the NaN values with the columns’ respective medians.
For “C”, which is also an unknown feature but categorical, we replaced the NaN values with “aa” which is the most frequently occurring value.
For “Country” we replaced the NaN values with “United States” because that’s where the overwhelming majority of individuals in the data appeared to be from.
It should be noted that a full and detailed explanation of our thought process is included in the notebook itself.

The feature that proved most difficult to work with was “stack_experience”, it is a feature that contains a list of codes that an applicant knows, the name of each code divided by a semicolon. Initially we thought a potential way to deal with this column’s missing values would be to rely on another feature, “is_dev”. Individuals who had “NaN” in “stack_experience” but “0” in “is_dev” would have the NaN value in stack_experience replaced with “ ” (a blank string). For individuals with “NaN” in “stack_experience” and “1” in “is_dev”, the process was a little more complicated. We would start by calculating the average number of programming languages an individual knows within the data, which would be done by counting the number of semicolons used to separate each programming language. Once we had the average, an additional column would be made that held the number of programming languages each individual knew. Individuals with “NaN” in “stack_experience” and “1” in “is_dev” would have the calculated average in “stack_in_num” (the column holding the number of programming languages an individual knows). In the case of having Nan in both “stack_experience” and “is_dev”, we first calculated the ratio of developers to non-developers using the given data, using this ratio we would randomly assign either “0” or “1” to each NaN value in the “is_dev” column. After which, we would apply the stack experience in accordance with the earlier strategies.
It is important to reiterate that we did not use the method stated earlier but rather that it was just an insight into the process. We chose to instead create a separate column by the name of “stack_in_num” which recorded the number of programming languages an individual knew. The numerical value in the “stack_in_num” column was recorded using a function that counted the number of semicolons used to separate each programming language. The NaN values were filled using Iterative Imputer.

In addition to filling in missing values, we also had to add new features to the data. Namely, convert categorical features into numerical representations. For every categorical feature, we created a dummy feature representing the values in the aforementioned categorical feature. For example, “worked_in_the_past” is a categorical variable that is either “T” if true (meaning an applicant worked in the past) and “F” otherwise. We converted the “T” values in the column to “1” and the “F” values to “0”. Likewise with “Disability”, “Is_dev”, and “Mental_issues”. For “Education”, “Sex”, “Age_Group”, and categorical feature “C” every unique value received its own column of dummy variables.
We also created a new feature “is_top_10” which is a dummy variable that would receive a “1” if the applicant was from a country among the top 10 countries that have the most job applicants for Hi-Tech positions, and “0” otherwise. It is important to distinguish that ”is_top_10” refers to the countries in our dataset and not globally. We initially considered making dummy variable columns for every country an applicant could be from in the data. Such a decision would have added too many new dimensions to the data and there was no way for us to know whether it would increase the models’ effectiveness; however it would undoubtedly increase the models’ computation time and complexity, which could result in an overfitted model. By creating the “is_top_10” feature we could not only effectively use the information in the “country” but also reduce the dimensionality of the data.
We also created a new feature “stack_in_num” which acts as a numerical representation of the “stack_experience” feature. We made a function that would count the number of semicolons used to separate each programming language in “stack_experience”. The result would be saved in a new column by the name of “stack_in_num”, which as a feature, represents the number of programming languages that an individual knows.

Once the missing values were replaced and the categorical features were dealt with, we could move forward and start properly creating the models. The first models that were created hadn’t undergone any tuning and the data hadn’t received any dimensionality reduction treatment. After their initial creation and first run, we focused on their optimization.

Starting with Logistic Regression,we noticed that the dimensionality of the data was an issue. Our model was underfitted, and so we chose not to apply any method that would reduce the dimensionality of the data for that model. The

reasoning behind this was that increasing the model complexity would help deal with the underfitting problem, however while the AUC score of the model improved the model was still underfitted. GridSearchCV was used to find the optimal hyper parameters.

In order to find the optimal hyperparameters for our Random Forest model we initially used the GridSearchCV function from the sklearn library. Grid Search Cross-Validation is a technique used in machine learning to search and find the optimal combination of hyperparameters for a given model. It systematically explores a predefined set of hyperparameter values, creating a "grid" of possible combinations. However, the computation time was too long. Instead, we switched to using gp_minimize from the scikit-optimize library. Gp_minimize is used to optimize the hyperparameters in a large and complex model. In order to use it, the user must first define a range where the optimal hyperparameter should be found. After which, gp_minimize would evaluate a user-specified number of random points within the range. The evaluations are recorded and the points are assigned values, the optimal function value and corresponding hyperparameter will be used as a benchmark for future iterations of the process. In each iteration, the function is used to find the hyperparameter value that influences the objective function in the desired way (Depending on the type of objective function, higher or lower values indicate a better fit), in comparison to the benchmark. The process continues until all user-specified iterations of Gaussian process regression are performed. In the evaluated hyperparameters the best hyperparameters will be the ones that cause the highest AUC score. The Random Forest model's AUC score was higher after using gp_minimize compared to when we tried GridSearchCV, and as an added bonus the computation time was significantly shorter than that of GridSearchCV.

In order to find the optimal hyperparameters for our K-nearest neighbor model, we initially thought to use gp_minimize as we did with Random Forest since it worked quite well. However the results were unsatisfactory, hence we elected to manually pick the hyperparameters. The final hyperparameters and the explanations are in the appendix.
The next step in optimizing our models was to perform dimension reduction. PCA with 99% variance was used on KNN and Multi-Layer Perceptron. This reduced the complexity of the data by removing the unnecessary features, which lowered the variance and improved the AUC score of these two models. However, PCA did not improve the Random Forest model so we tried using RFE. RFE differs from PCA in the sense that it selects a subset of the original features without transforming them. It removes features recursively based on their importance for predicting the target variable. While RFE selects features that are useful for a specific prediction task, PCA transforms features more generally to uncover latent characteristics. That being said, using the RFE method on the Random Forest model produced a better AUC score than when we used PCA. In addition, we removed the anonymous feature "B" as it had a high correlation with years of experience (0.85), this improved the AUC score of the Random Forest but not any of the other models. As for the Logistic Regression model, as stated earlier, we chose not to undergo any form of dimensionality reduction. This is because a more complex data set can be used to combat underfitting, which was a problem in our Logistic Regression.

After the creation of the fourth model, which was the Multi Layer Perceptron, was when we realized that the data had not been split correctly and the models were trained on both the training set and the validation set. This was an issue for obvious reasons, the model results were not accurate because the models were being "validated" on data they had seen before. Once the issue was rectified and it was ensured that the data was split properly, and the models were not trained on the validation sets, we ran the models again. This time every model except the Random Forest produced an AUC of 0.5. This meant we had to find new ways of optimizing the models.

The first thing we decided to do was to search and remove outliers in the data. This entailed visualizing the features using box-plots and histograms. In an attempt to deal with the outliers in prev_salary, we first defined what an outlier would look like. Since salaries in tech positions vary by country, it was decided that an outlier would be a salary that was lower than the salary in the lowest paying country. According to an article by the Mauve Group titled "The world's highest and lowest tech salaries", published on March 12, 2024, the lowest paying high tech salary is in India and stands at $7,225 USD annually. However, this proved to eliminate a large portion of the data so we settled on defining any salary under $5000 USD as an outlier instead. We initially wanted to set a higher bound as well for salaries that exceeded that of the highest paying country, which is the United States with an average salary of approximately $110,000 USD annually. However, with both bounds active, they eliminated 20% of our data. Hence, we decided to not add an upper limit.
In order to find outliers in the anonymous features "A" and "B"we used a box plot graph to visualize the data. According to the graphs, the values greater than 20 in "A" are outliers and the values greater than 25 in "B" are also outliers. In "A" all values greater than 30 or less than -16 were removed, and in B all outliers greater than 40 were removed.

Additionally, we checked for outliers in the "country" feature. While the number of applicants for HighTech jobs vary greatly by country, the ratio of those who are accepted to those who are rejected seems relatively consistent. Therefore we didn't remove any values there.

Initially, we attempted to scale the data using Min Max scaling on the whole data. However, the AUC scores did not improve. The next thing we tried was Robust Scaling. For the purpose of the KNN model, when the data is not scaled down some values will appear larger than others which creates a "bias". This bias causes the model to output inaccurate predictions. The distance between each neighbor might seem too big. In reality, the proportions are just different in those features, hence the big distance difference. Robust Scaling also helped us with Logistic Regression and the MLP. Although Logistic Regression doesn't require the data to be normalized in theory, the model didn't perform well until we applied the Robust Scaling on the data it was training on. MLP on the other hand is much more sensitive to different scales between the features, because features with larger scales tend to dominate the gradient updates, leading to inaccurate and bad model performance. Robust scaling improved the AUC score of every model other than the Random Forest. After scaling the data, we ran the gp_minimize function again in order to find new hyperparameters but there was no difference from the previous iteration.

Once everything was finalized and we compared the results of all the models, we decided to choose the Random Forest Classifier model as it had the best AUC score.

Appendix

Random Forest

1. N_estimators- defines the number of trees to be used in the model, a higher number of trees is usually indicative of a more accurate model. However, more trees increases the time it takes to train the model.

   In our model we chose 500 as the number of estimators. The main reason is because it was recommended by the gp_minimize function, but also because this number of trees contributes to a higher auc score without having too lengthy of a computation time.
2. Max_depth- the maximum number of splits a tree can take, or in layman's terms how many levels the tree has. If the max_depth is too low, the model will be trained less and have a high bias, leading the model to underfit. In the same way, if the max_depth is high it could lead the model to overfit.

   In our model we chose 38 as the maximum depth, as recommended by the gp_mnimize function. The max depth was chosen because a larger depth wouldn't have improved the model results without causing overfitting.
3. Min_samples_leaf - The minimum number of samples required to be at a leaf node. Regardless of the depth, a split point will be created if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

   In our model we chose 5 as the minimum number of samples, as recommended by the gp_mnimize function. Increasing the value in min_samples_leaf has a similar effect to increasing the max_depth feature, hence setting a larger number of samples as the minimum would likely lead to overfitting.
4. Min_sample_split- determines the minimum number of decision tree observations in any given node in order to split.

   In our model we chose 20 as the minimum number of samples, as recommended by the gp_mnimize function. By setting a minimum number of samples required to split a node, we can prevent the tree from overfitting by creating branches with very few samples.
5. Criterion- The function to measure the quality of a split

   In our model we chose "entropy" as the criterion, it is a measure of randomness or unpredictability in the data set. The entropy is calculated using the following formula:

   $$\text{Entropy} = -\sum_j p_j \cdot \log_2 \cdot p_j$$

   pj is the probability of class j.

   Entropy is a measure of information that indicates the disorder of the features with the target. The optimum split is chosen by the feature with less entropy. It gets its maximum value when the probability of the two classes is the same and a node is pure when the entropy has its minimum value, which is 0. We chose this

criterion rather than Gini because gp_minimizer recommended doing so, which makes sense because "entropy" yields better results on average than "gini" although not by much.

KNN

1. N_neighbors- the number of neighbors to reference in kneighbors, the number of neighbors required for each sample. A larger k means a less complex model. On the other hand, a smaller k means a more complex model, which can lead to overfitting.

   In our model we chose to set the number of neighbors as 10, as recommended by the gp_mnimize function.
2. Weights- the weight function will only be used after the model has been trained in order to output the prediction, there are a few settings to choose from, each impact the final prediction of the model in different ways. 'uniform' : uniform weights. All points in each neighborhood are weighted equally. 'Distance' : weight points by the inverse of their distance, in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away. [Callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

   In our model we set the weights parameter to "distance". The "distance" weight reduces the bias by down-weighting data points that are less similar, but by doing that it increases the variance since the prediction relies more on individual data points of the training sample.
3. Algorithm (Brute)- Algorithm used to compute the nearest neighbors.

   In our model we chose the "Brute" algorithm. Brute Force KNN algorithm calculates the squared distances from each query feature vector to each reference feature vector in the training data set. Then, for each query feature vector it selects k objects from the training set that are closest to that query feature vector.
4. Leaf_size- The leaf size controls the minimum number of points in a given node, and effectively adjusts the tradeoff between the cost of node traversal and the cost of a brute-force distance estimate. A higher leaf size usually results in query time reduction. The larger leaf_size, the closer neighbors the algorithm picks.

   In our model we chose a leaf size of 6
5. P (Euclidian)- Power parameter for the Minkowski metric. Minkowski Distance is a metric intended for real-valued vector spaces. We can calculate Minkowski distance only in a normed vector space, which means in a space where distances can be represented as a vector that has a length and the lengths cannot be negative.

   The P value we chose for our model is p = 2. Euclidean distance is the straight-line distance between two points in a vector space. It is calculated by taking the square root of the sum of the squared differences between the corresponding coordinates of the two points. It is the most commonly used distance metric in KNN, as well as the most intuitive.

Logistic Regression

1. Penalty - Penalty is calculated as the square root of the sum of the squared vector values. This term will help shrink the coefficients in the regression towards zero. The penalty intends to reduce model generalization error, and is meant to disincentivize and regulate overfitting. Technique discourages learning a more complex model, so as to avoid the risk of overfitting.

   In our model we chose "l2", as recommended by the gp_minimize function. The l2 penalty is the sum of the squared values of the model coefficients multiplied by a regularization parameter, which controls the strength of the penalty.

2. max_iter - Maximum number of iterations taken for the solvers to converge. In logistic regression, having more iterations is generally better. However, there is a trade-off as the more iterations the model has to execute, the more its computation time increases.
    a. Our model executes 100 iterations, as recommended by the gp_minimize function. This is also the default number of iterations set by the LogisticRegression function.
3. C- C is also known as the regularization strength, regularization strength works with the penalty to regulate overfitting. Smaller values specify stronger regularization and high value tells the model to give high weight to the training data. It must be a positive number.

    In our model c is set to 0.046415888336127774, as calculated by the gp_minimize function. Our C is lower than '1' which is the default value. This means that our model does not assign a lot of weight to the training data.
4. Solver - the solver is an algorithm or method that the model uses to find the minimum cost, or the smallest loss.

    Our model uses the "liblinear" solver. It uses a coordinate descent algorithm. Coordinate descent is based on minimizing a multivariate function by solving univariate optimization problems in a loop. In other words, it moves toward the minimum in one direction at a time. Liblinear is generally better suited for small datasets, and can only handle binary classification by default.

<u>Neural Network</u>

1. Hidden_layer_sizes- A higher number of hidden layers can allow the model to learn more complex relationships in the data but also increases the risk of overfitting. Generally, a larger dataset and a more complex problem require a deeper network with more hidden layers.

    In our model we have 2 hidden layers, as recommended by the gp_mnimize function and 23 features after pca. The first layer size is 113 and the second layer size is 21.
2. Solver- the solver optimizes the model by coordinating the model's interface and backward gradients to form parameter updates in an attempt to minimize loss. The solver oversees optimization and generates the parameter updates as a form of learning.

    LBFGS is an optimization algorithm that does not use a learning rate. It is based on the assumption that the function that needs to be optimized can be approximated locally The process looks like this: The function starts with an initial guess, from there it uses the jacobian matrix to compute the direction of the steepest descent, the next step is to use the hessian matrix to compute the descent step and reach the next point. This process repeats until convergence. LBFGS works best for smaller datasets as it can converge faster than the other solvers.
3. Activation - Activation functions are used to introduce nonlinearity to models, which allows deep learning models to learn nonlinear prediction boundaries.

    In our model we used "relu" as the activation function for the hidden layer, as recommended by the gp_mnimize function. The rectified linear unit (ReLU) introduces the property of nonlinearity to a deep learning model and solves the vanishing gradients issue. If a model input is positive, the ReLU function outputs the same value. If a model input is negative, the ReLU function outputs zero.The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time. It is one of the most popular activation functions in deep learning.
4. alpha - Strength of the L2 regularization term. Alpha is a parameter for regularization term, or penalty term, that combats overfitting by constraining the size of the weights. Increasing alpha may fix high variance ,which is a sign of overfitting, by encouraging smaller weights. This results in a decision boundary plot that appears with lesser curvatures. Similarly, decreasing alpha may fix high bias (a sign of underfitting) by encouraging larger weights, potentially resulting in a more complicated decision boundary.

    In our model the alpha is set to 0.018153987421074984, as recommended by the gp_mnimize function. Since the alpha is less than 1, our model aims to fix its underfitting issue by raising model complexity.
5. batch_size - batch size is the number of training samples given to the network after which parameter update happens. A large batch size basically means that the model goes through more samples before the parameter update. It can lead to more stable learning but requires more memory. A smaller batch size on the other hand updates the model more frequently.The value of the batch size affects memory and processing time for learning.

The batch size set in our model is 103, as recommended by gp_minimize. It is a relatively large batch size, which makes sense when considering the dimensions of the data is also quite large. Therefore it is optimal for the model to sift through more samples per iteration.

6. learning_rate - The learning rate defines how quickly a network updates its parameters. If the learning rate is high, the model learns quickly but might make mistakes. If the learning rate is low, the model learns slowly but more carefully. This leads to less errors and better accuracy, however takes a lot of time which can be expensive for large data sets.

   The learning rate we chose for our model is "constant", as recommended by the gp_minimize function. Initially, each parameter is assumed or assigned random values. A cost function is generated using the initial values, and the parameter estimations are improved over time to minimize the cost function.

7. max_iter - The maximum number of iterations the solver executes. Having too large of a number can lead to overfitting however, a low number can not comprehend complex data.

   In our model we chose the max number of iterations to be 438, as chosen by the gp_minimize function. The function likely deduced that setting a higher number of iterations would not benefit the overall model performance.

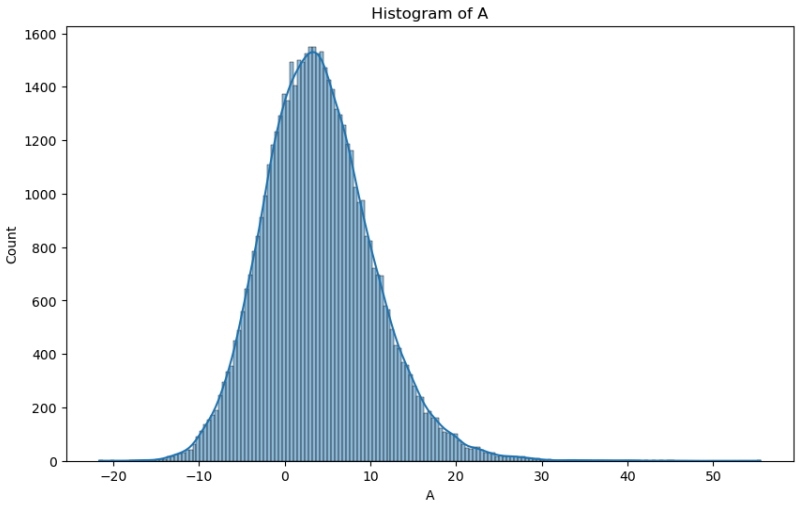
Responsibility Log

Netanel Druyan - 212356661 - Wrote the code for the preprocessing and the 4 models. Looked for the best hyper parameters and features, created all the plots, correlation matrix, confusion matrix and provided explanations. Created the pipeline, and the final predictions.

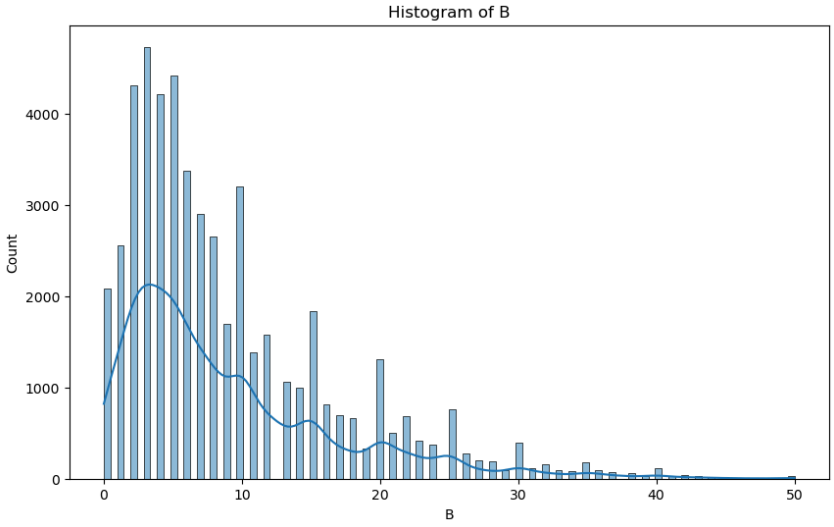
Naama Tanchuma 322432071- Handled around 20% of the data analysis, Wrote the summary report, wrote detailed explanations for the questions in part 2, wrote explanations for the visuals, made the dummy variables and added them to the dataset, did graph analysis, wrote detailed explanations for feature importance.

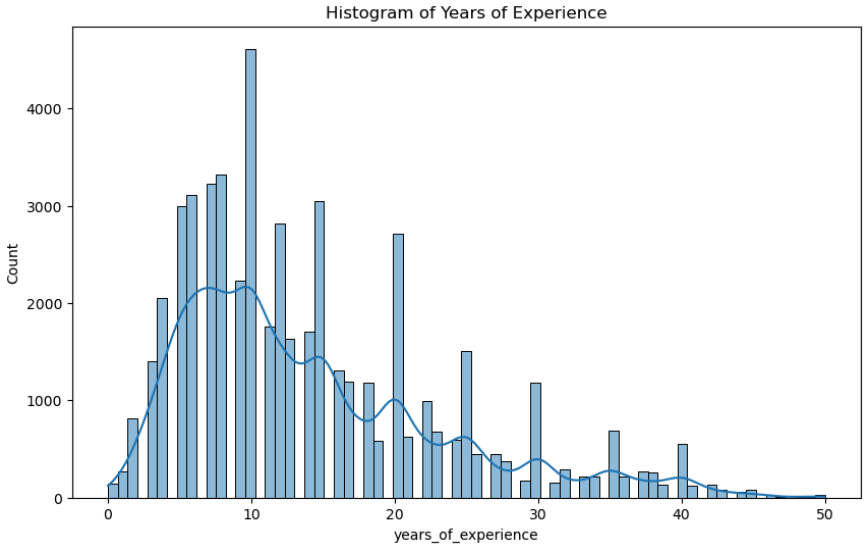
Shared responsibilities- filling in the NaN values, finding ways of dealing with categorical variables, researching ways of optimizing the models.

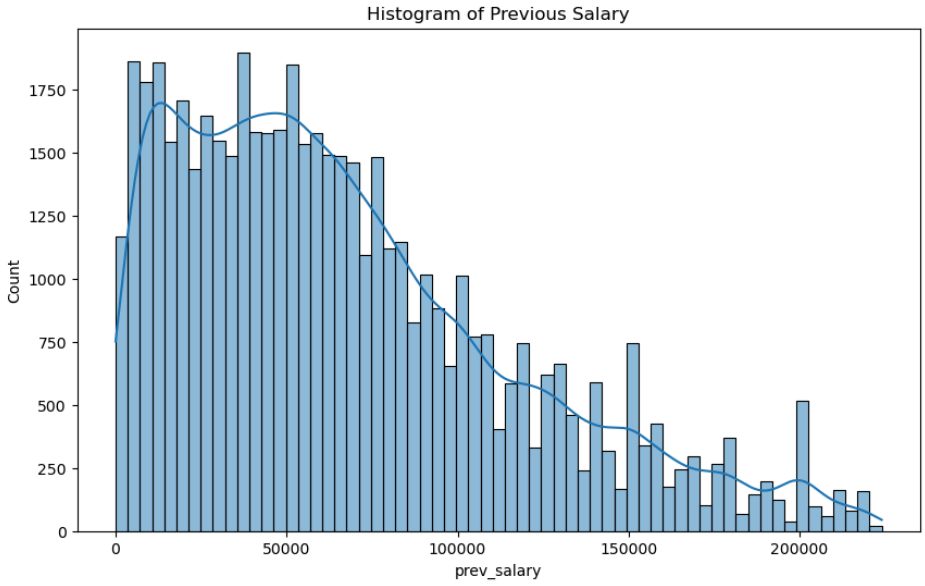
Graphs (analysis appears in notebook)

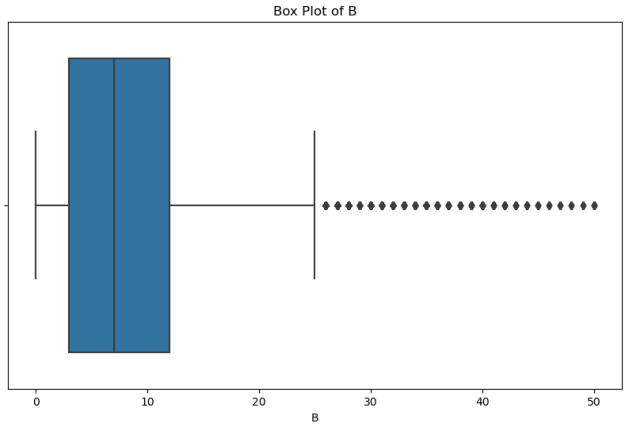


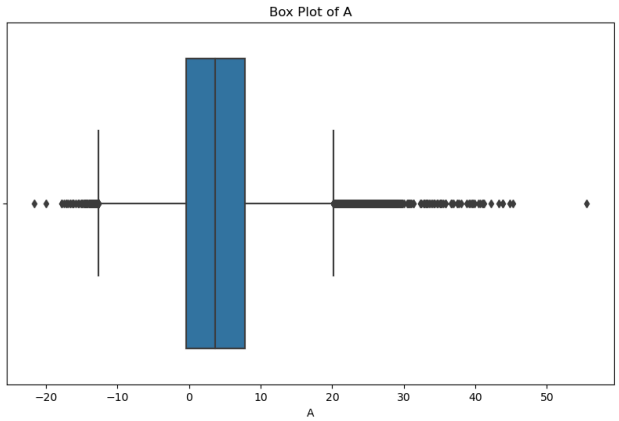
We can see that the anonymous feature “A” is the only feature that follows a normal distribution.



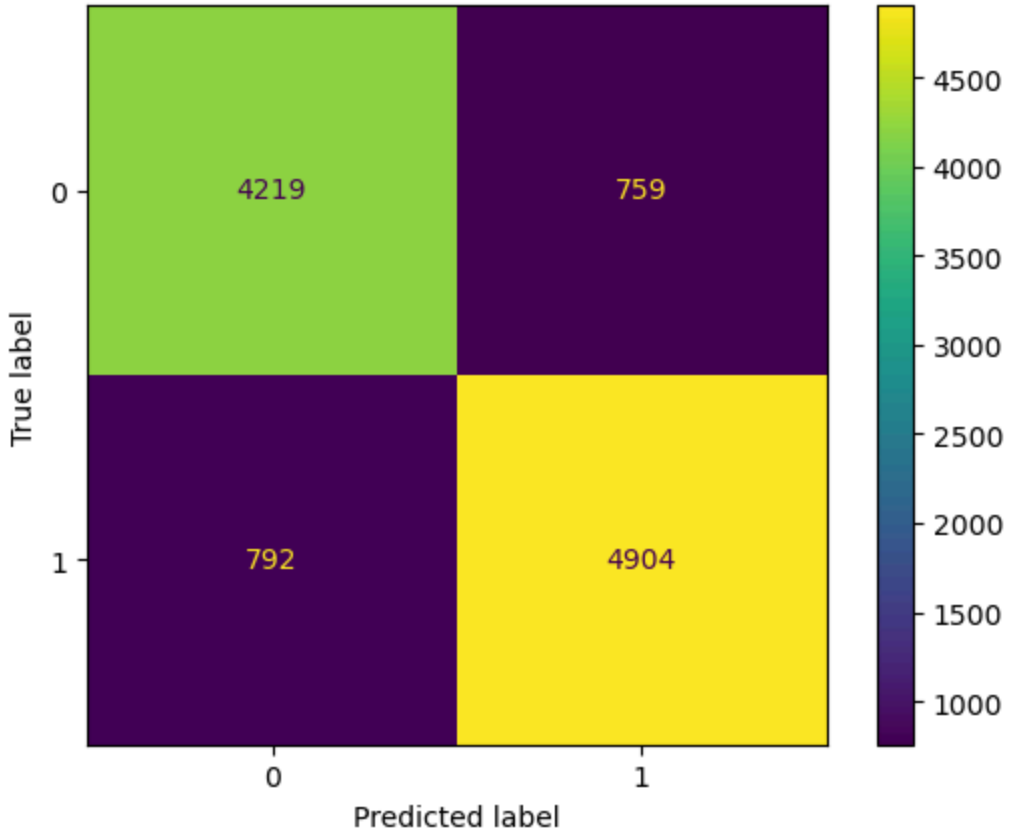






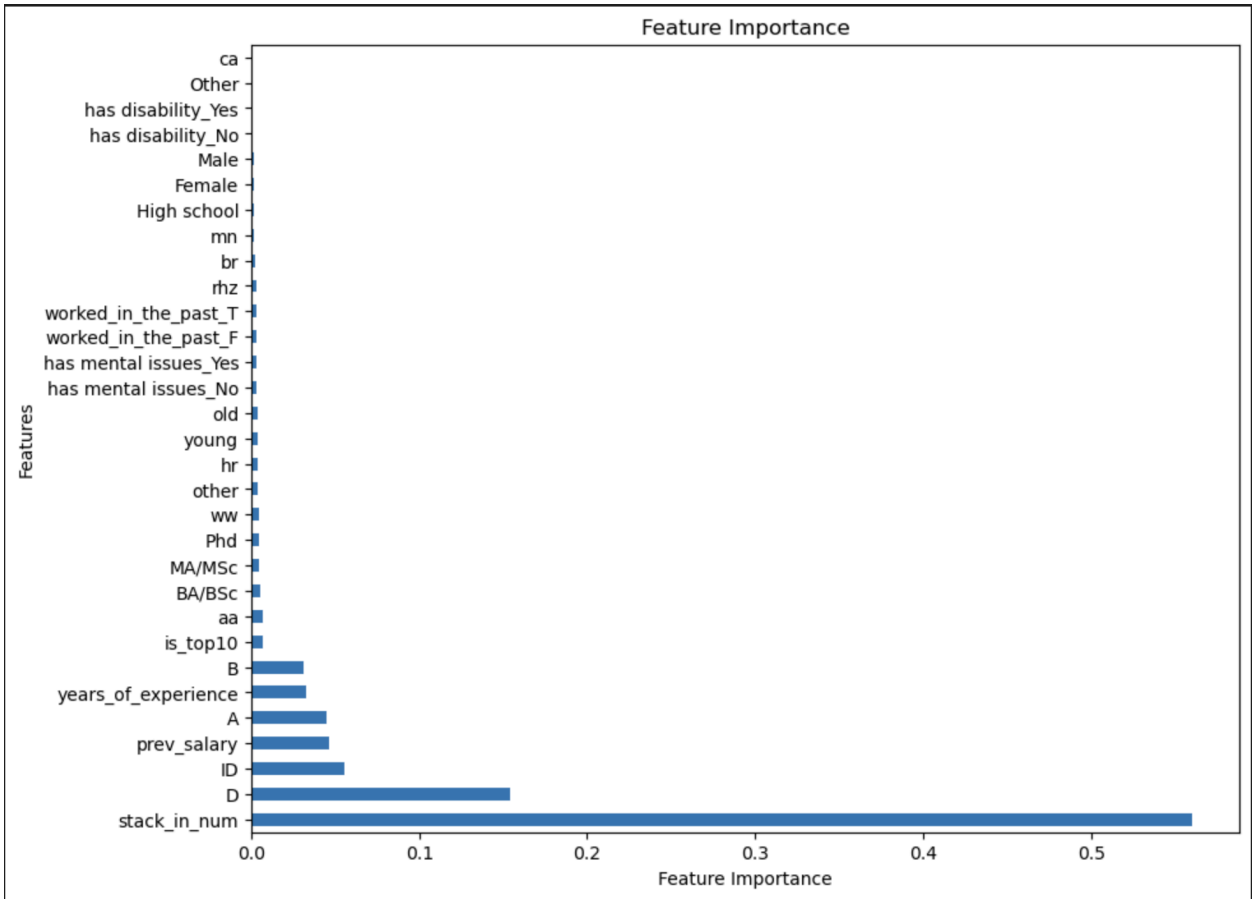


Confusion Matrix

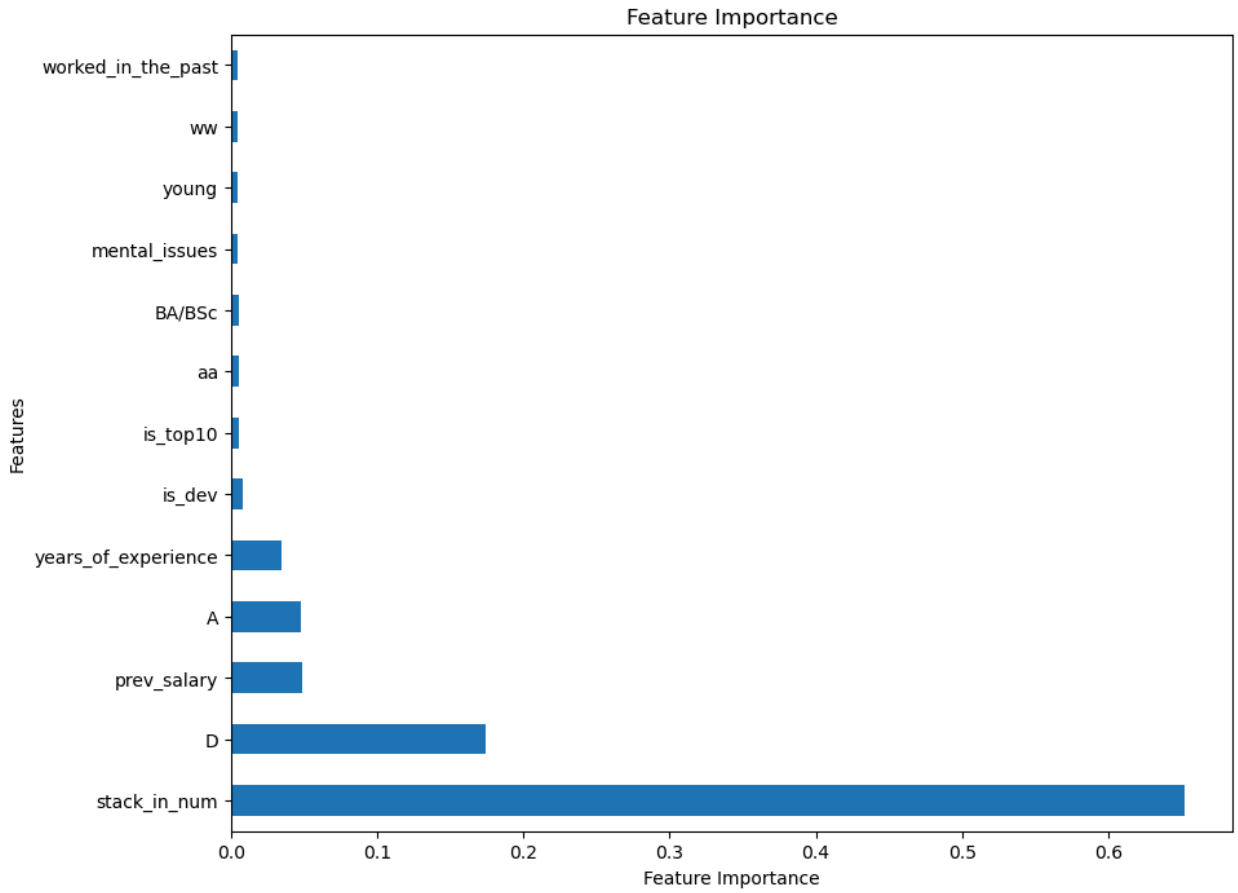


The confusion matrix gives us insight into the overall accuracy of the mode.
The accuracy of the model is: 85.46%
The precision of the positive class: 84.75%
Recall of the positive class: 84.19%
Specificity: 86.59%
The model seems to classify false positives and false negatives at about the same rate.

Feature Importance, with cross validation, without pca

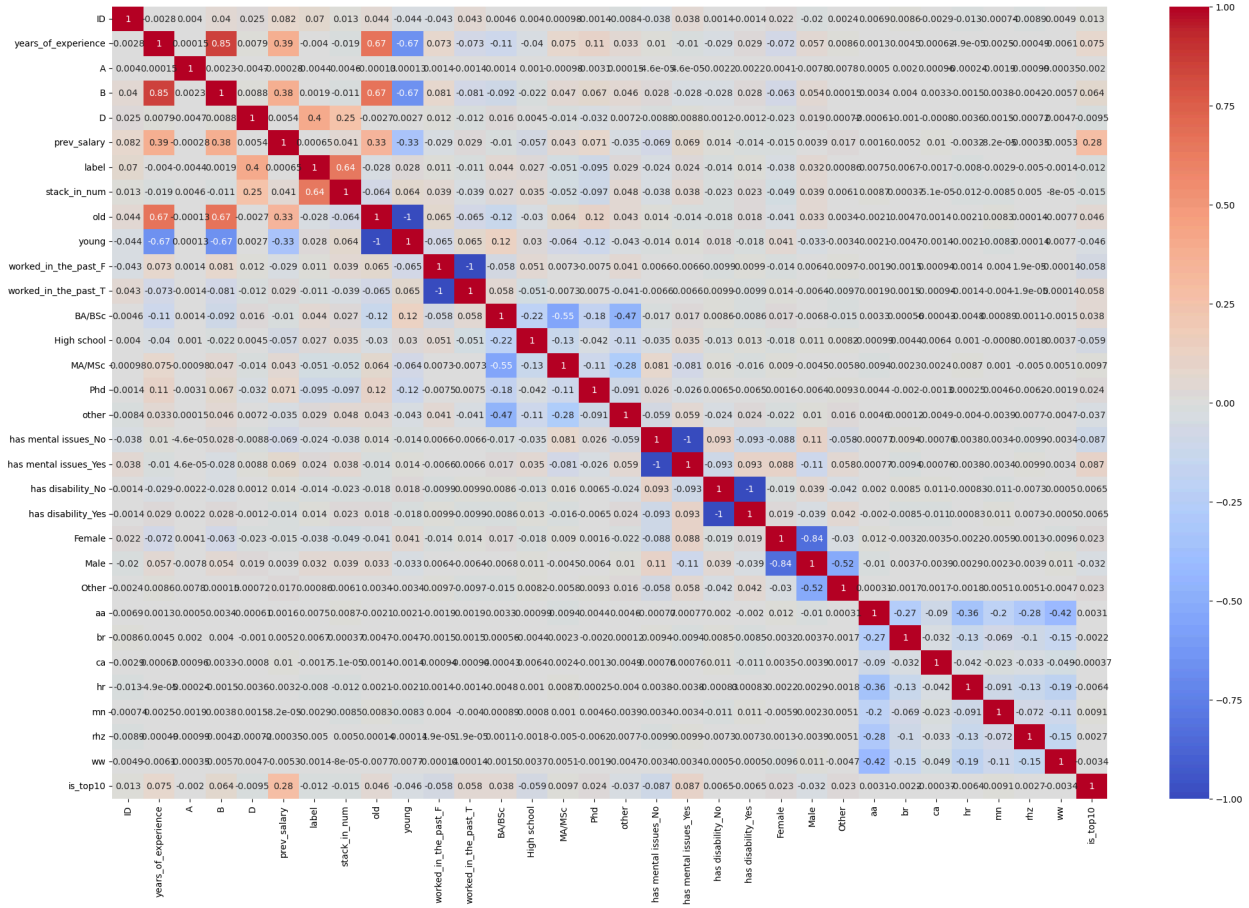


We accidentally included ID in the list of relevant features, it came out as third most important. Other than that, the graph shows that the most important feature in determining the likelihood of an applicant being accepted to work is the number of coding languages they know (we can see that stack_in_num is the largest bar and exceeds 50%). Following coding languages, we see that the most relevant features are anonymous feature D, which we can't make any deductions from because we don't know what it is, and the applicant's previous salary. A possible explanation as to why the previous salary of an applicant is given so much importance is that it correlates to the individual's past job experience and country of residence. To clarify, the annual salary of a tech worker varies greatly in each country. For example, an individual with a previous salary exceeding $100,000 USD is likely to be from the United States, where we observed a large acceptance rate of individuals who apply for HighTech jobs. Alternatively, an individual with a high previous salary was perchance employed in a high paying position, making them a more credible candidate which in turn would influence their chance to be accepted to work in a HighTech position.



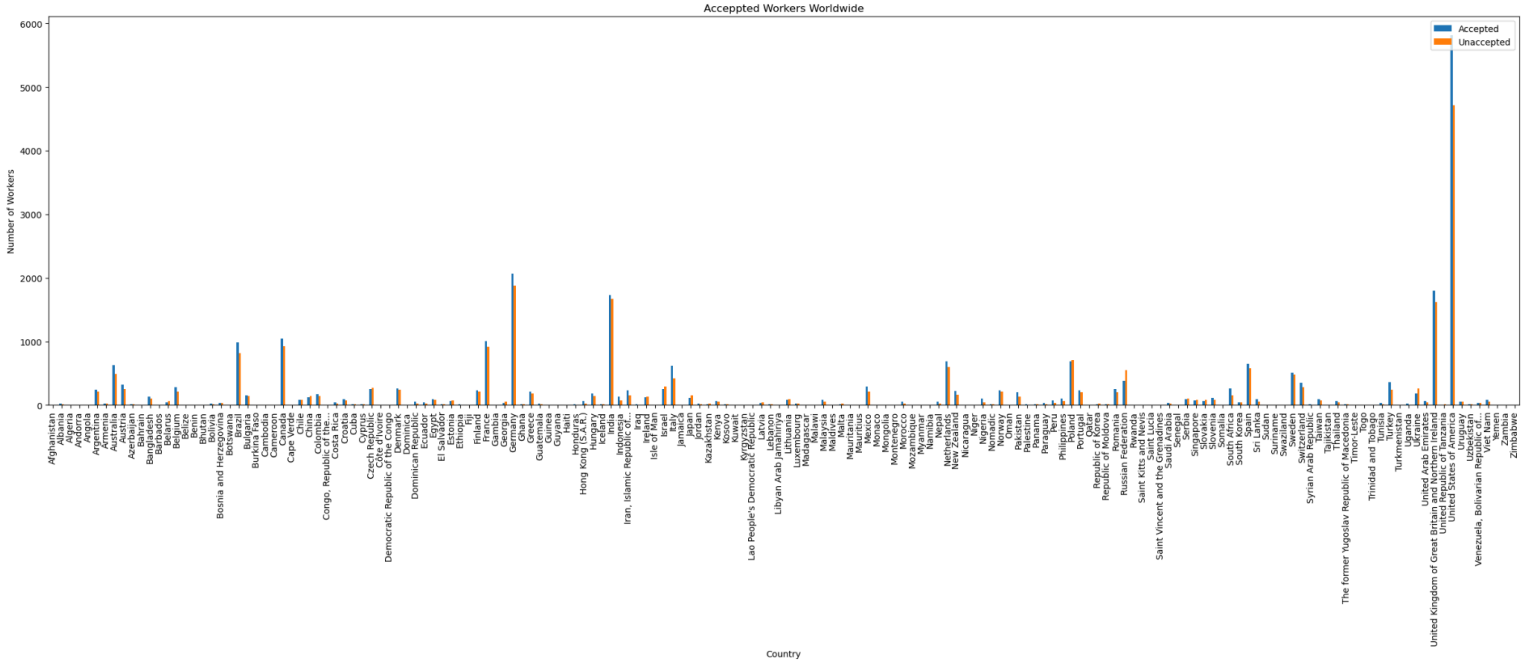
This is the same feature importance graph, after PCA and normalization. Not much has changed, but we can observe that stack_in_num now has been attributed more importance as it now sits at over 60%. None of the other features that were left after the dimension reduction received any significant change, like that of stack_in_num. From this we can infer that the unnormalized data was reliable.

Correlation Matrix of the features, before applying PCA

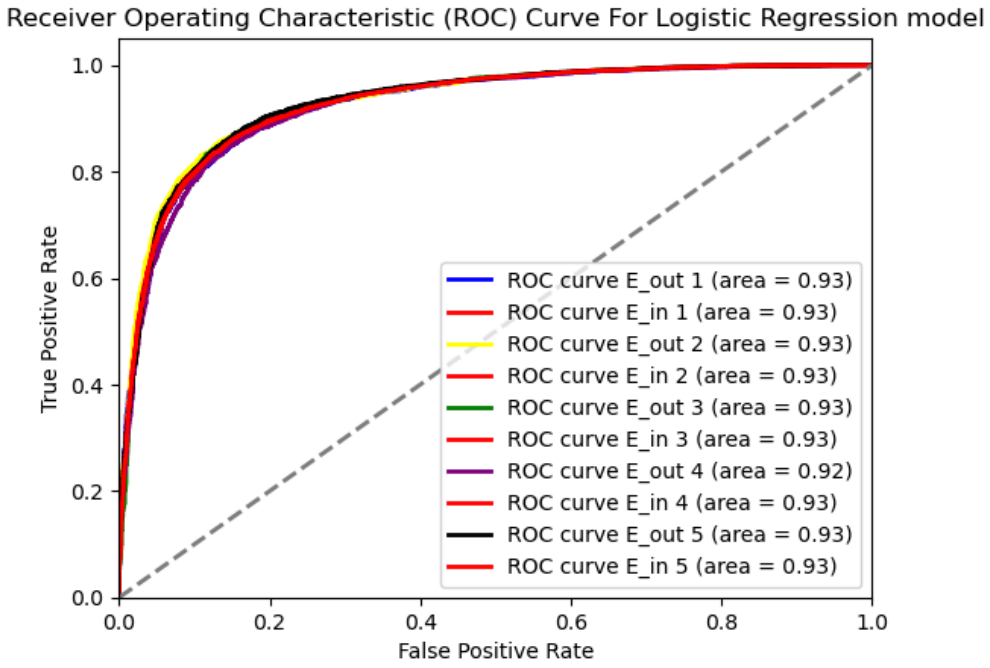


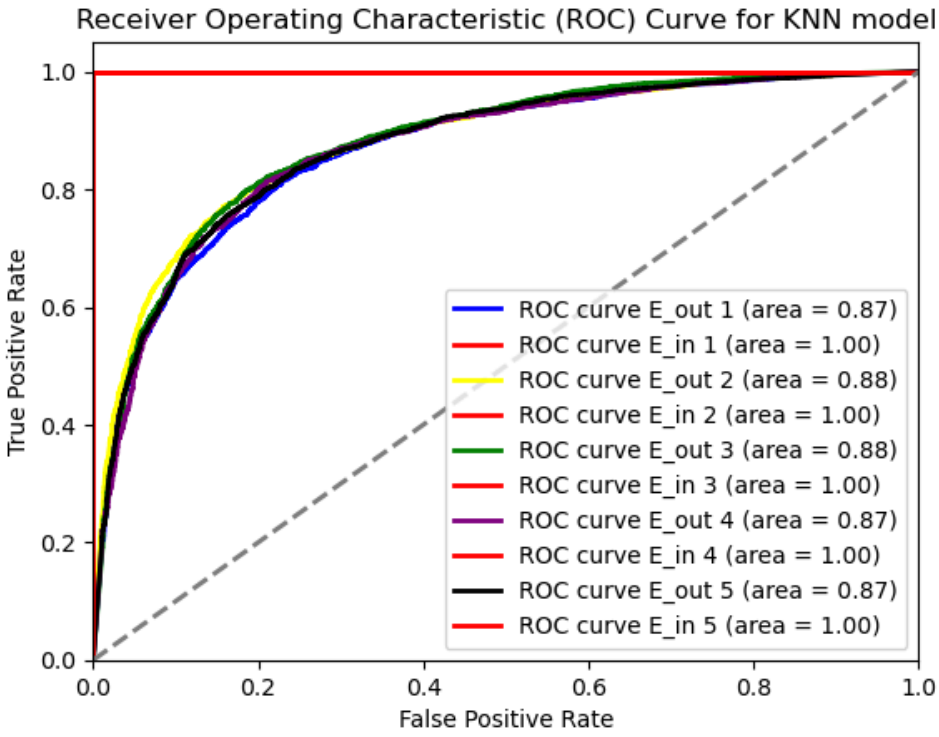
We can see a surprisingly high positive correlation between anonymous feature“B” and “years of experience”, and that “young” has a high negative correlation with “B” and “years of experience”. The negative correlation between “young” and “years of experience” is expected, as someone who hasn’t had much life experience, wouldn’t have much work experience either.

Histogram of the number of individuals in the data divided by countries, and whether they were accepted for the job

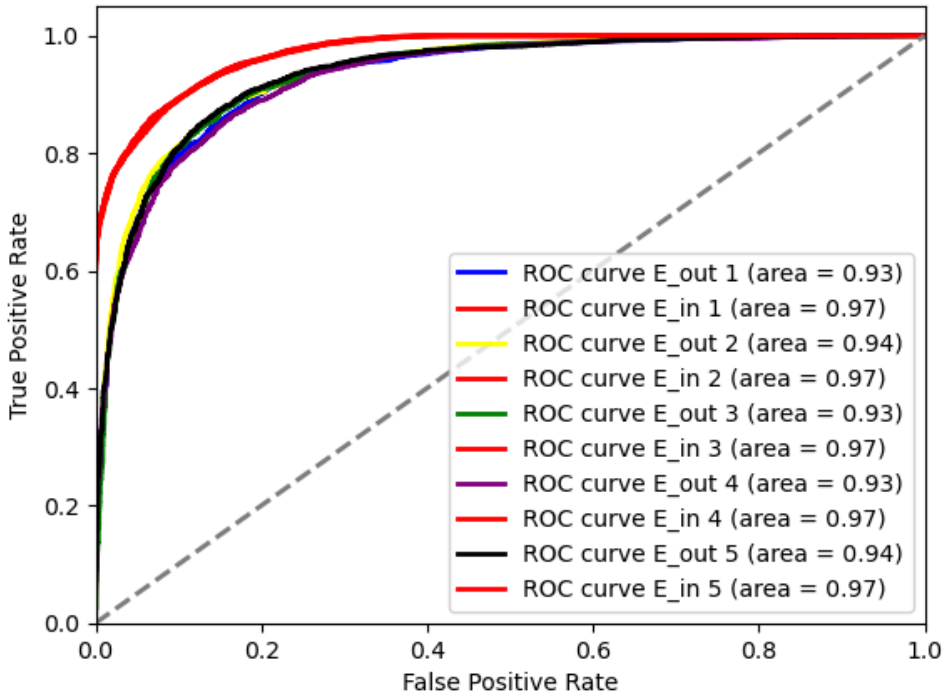


Logistic regression, with cross validation, without PCA (to increase complexity)





Receiver Operating Characteristic (ROC) Curve for Random Forest model



Receiver Operating Characteristic (ROC) Curve for Multi Layer Perceptron model