Saira Poonnen

CS4641 Final Project

The United States, despite being a developed country, has great levels of income inequality. There is a high poverty in the United States of America where in 1 in 5 kids are in poverty [1]. As an international student in the US, I was intrigued by this level of inequality and wanted to delve further into factors that affect it. In one of my courses this semester, I learned about the effects of factors such as race, education, sex and social class on income. In this project, I will explore the effects of these factors to make predictions on income. In doing so, I can make conclusions on what factors are necessary to improve one's level of income. I will use supervised algorithms namely Random Forests with Bagging, Support Vector Machines and Neural Networks.

The dataset I will be using for this is the Census Income dataset from the UCI Machine Learning repository [2]. This dataset has 48,842 instances with 14 attributes from a 1994 Census database. These attributes are both categorical and numerical namely:

- age
- workclass
- fnlwgt
- education
- education-num
- marital-status
- occupation
- relationship
- race
- sex
- capital-gain
- capital-loss
- hours-per-week
- native-country

The class label is a binary value which represents whether the income is greater than $50,000 per year or less than/equal to $50,000 per year. Through this we can predict based on all these factors if a person's annual income is greater than $50,000.

Data Pre-processing

The UCI website provides the training and testing data already split so I will use as my training and testing data respectively. The training dataset has 32561 instances (approximately 2/3 of the dataset) and the testing dataset has 16281 instances which is approximately 1/3 of the whole dataset which was split randomly. The UCI website states that there are missing values in the dataset. On looking through the dataset I see that there are "?" in some of the columns for native-country, occupation and workplace. There are 2399 rows that have missing values in the training dataset and 1221 rows that have missing values in the testing dataset which is a total of 3,620 missing values. Since the missing rows only make 7.4% of the data, we are still left with 45,222 instances making it a non-trivial

dataset. So, I removed these rows from my dataset as there will not be as much information lost. I will also remove the column fnlwgt as this is an estimated number of people that a specific entry can represent and will not affect classification of income of a specific entry.

The native-country attribute had 41 different countries. To reduce this, I grouped the countries into specific regions as asiaEast = {" Cambodia", " China", " Hong", " Japan", " Laos", " Philippines", " Thailand", " Taiwan", " Vietnam"}, middleEast = {" Iran"}, asiaSouth = {" India"}, latinAmerica = {" Columbia", " Cuba", " Ecuador", " Guatemala", " Jamaica", " Nicaragua", " Puerto-Rico", " Dominican-Republic", " El-Salvador", " Haiti", " Honduras", " Mexico", " Peru", " Trinadad&Tobago"}, europe = {" Greece", " Italy", " Portugal", " Germany", " Poland", " Yugoslavia", " Hungary"," England", " Holand-Netherlands", " Ireland", " France", " Scotland"}, us = {" Outlying-US(Guam-USVI-etc)", " United-States"}, canada = { " Canada"}, misc = {" South"}. There will not be much information lost as most of the instances (43,832 instances) have US as the native country. By grouping by regions, I will be able to preserve the information based on geographical location but reduce the number of overall dimensions to avoid affecting the computational complexity and reduce risk of overfitting with too many different categories to fit the model to.

Since there were 8 categorical attributes and some algorithms such as Support Vector Machines require numeric forms of data, I used One-Hot Encoding from sklearn [3] library to convert the data which has an advantage over Label-Encoding as it does not assume that the number associated with a category has a weightage on the importance of a category. For each categorical feature, this will create a new column for each different category where only one column has 1 and the remaining 0 depending on which category the attribute falls in.

The metrics I will use to measure performance are accuracy and F1 score. Accuracy represents classification rate as it states a ratio of how many true positive and true negatives there are compared to the total number of values. It provides a value to indicate how many times the classifier provided the right prediction. I chose to use accuracy even though there is class imbalance because the imbalance is not too high as to decrease significance of accuracy. Since there was class imbalance, I will use an additional metric as well of F1 score. This is suitable for models with imbalanced classes and for binary classification scores. This score is a harmonic mean between precision and recall. Precision is a ratio of true positive to true positive and false positive. Recall is a ratio of true positive to true positive and false negative. F1 score will allow us to consider both precision and recall and provide information as to getting correct predictions without missing too many.

Algorithm Descriptions

Random Forests

Random forests are a classification algorithm that consist of many decorrelated decision trees. Each decision tree predicts a specific label/outcome and the mode outcome is taken as the prediction of the random forest. In this way we can improve the generalization of decision trees and reduce overfitting to a specific dataset. We use bagging and create random samples that we build our decision trees based on. We randomly sample the data into the number of trees needed without replacement,[4] so we train the model on a subset of the training dataset. We set bootstrapping to True and this will allow us to decrease variance and improve generalization.

The parameters I selected to tune are:

- n_estimators: This represents the number of trees in the forest. If the number of trees is increased, then we reduce the risk of overfitting and our model is more generalizable. We reduce the bias from the training data. However, this would be computationally expensive, so we need to find a balance between a value too high or too low. The complexity of the model will increase with an increased number of estimators. The default value for this is 100.
- max_features: This is the number of features we want to consider when making a split. Increasing this value increases the complexity of the model as we have to consider more features and fit model specifically to the training dataset. This could lead to overfitting to the training data and affect the generalization of the model.

Support Vector Machine

Support vector machine is a supervised machine learning algorithm that takes in datapoints and finds the best hyperplane that separates the different classes. The best hyperplane is one that maximizes distance to training data. This will ensure that our model is less affected by noise in the training data that may lead to incorrect classification when tested. The points that determine this decision boundary are the support vectors and these are the points that give the best margin so not all the data points have an impact on the decision surface. This algorithm can also be used on data that is not linearly separable as we can project to some higher dimension in order to improve the linear separability. Kernels can be used for this feature transformation and help classify non-linearly separable data. The hyperparameters I will be tuning are:

- C: This is the regularization parameter. This is the penalty to misclassifying training points. Increasing the value of C can increase overfitting and thus reduce the generalizability. There will be higher variance and lower bias so we need to balance the value. The complexity of the decision surface will increase as the value of C increases as we fit to the training data more closely.
- Gamma: This is the kernel coefficient. The higher the value is the more complex the model is as we will fit the training data more closely as the decision boundary shape is affected. This will increase risk of overfitting if value becomes too high.

I did not choose kernel as one of the hyperparameters as the RBF kernel with certain C and gamma parameters performs similarly to the sigmoid and linear kernel. The polynomial kernel also has many hyperparameters which would need to be tuned so I chose to use the RBF kernel [5]

Neural Network

A neural network is a classification algorithm that is composed of perceptrons, similar to how the human brain is composed of neurons. The model is composed of an input layer of neurons, hidden layers and then an output layer. Each neuron in a layer gets input from all the neurons in the previous layer and this is weighted and passed through an activation function, such as the sigmoid function, and then passed on to the next layer modeled after the human brain neurons passing signals [6]. There are specific iterations called epochs and at each epoch the model is re-evaluated and improved by adapting the weight vector so that classification can be improved. I did not consider number of hidden layers or units in the layer in the hyperparameters as I considered this a parameter determining how the algorithm is structured rather than how it was fit. I used Keras [7] to build the Neural Network. The architecture I used is 4 dense layers with outputs of size 32, 16, 4 and 1 respectively. I used the default activation function of relu for each of the layers except sigmoid for the last layer as it will provide a probability for each class which I can then use to classify. I used a loss of 'binary_crossentropy' since I was doing a binary classification.

The hyperparameters I will be tuning are:

- Learning Rate: Neural networks utilize gradient descent which involves this parameter. This controls how quickly convergence occurs. If it is too low, then we need many iterations to converge. However, if it is too high, we risk overshooting and diverging from the minimum loss and may never find the minimum loss. Default learning rate is 0.01
- Optimizer: This determines what optimizer is used to minimize the loss function.

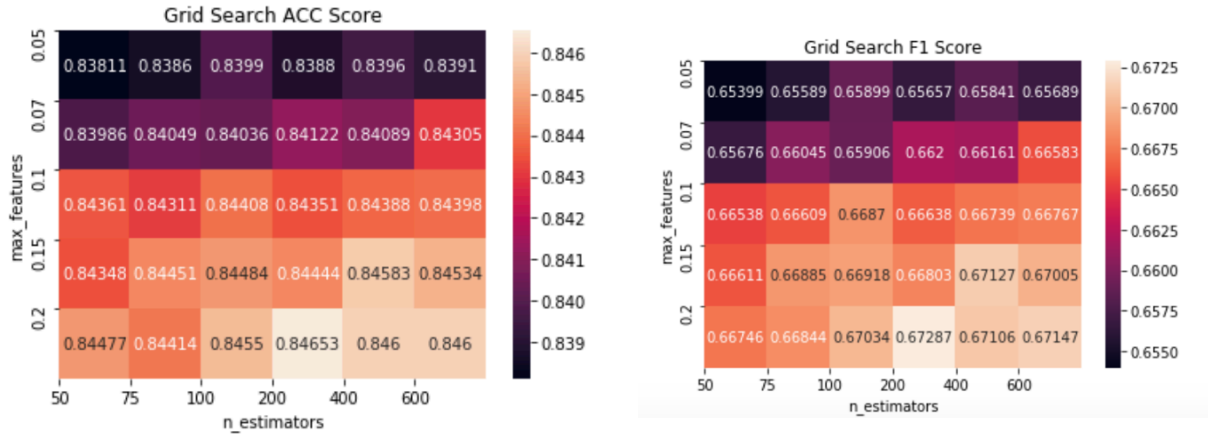
I chose to tune these hyperparameters as I believed they will have an impact on the resulting model and affect the performance metrics and were the most commonly tuned parameters. I used the default values for all the other hyperparameters to reduce the overall computational cost in choosing the best hyperparameters and then training the dataset.

Tuning Hyperparameters

There are a number of hyperparameters for each algorithm. In the previous section I discussed which hyperparameters I will be tuning for each algorithm while using the default values for the remaining so that I can compare the results and choose the best values. To do this I will use GridSearchCV [8]. Using this, I pass in a dictionary of the parameters I will test and the metrics I will use to assess performance namely: f1 and accuracy score. I will use 3-fold cross validation so that the training set is split into three sets and one set is held out as the validation set. The model is trained on the remaining two sets and tested on the validation set. This process is repeated until all three sets have been the validation set at a specific iteration and the mean score is taken.

By using cross validation, I will reduce the risk of overfitting too closely to the training data as I will use scores from the validation set which was unseen while training the model to see which hyperparameters produce good results. This will also help choose hyperparameters that will create a more generalizable model so that it can perform well on unseen data. I will also use 3-fold cross validation on only my training data so that I can assess performance based on unseen data. The reason I am using 3-fold cross validation over just cross validation is because I can reduce the variance between the different splits by averaging the scores from the different splits and improve the precision. In this manner, where the data is split will have less of an effect on the resulting model. I am using 3 folds as my training data is large (32561 instances), so I aim to keep the computational complexity low for this step. It is also important to not include the testing data in this tuning process so that performance can be assessed on new data after choosing the best parameters for this specific problem and model.

Random Forest Hyperparameter Tuning

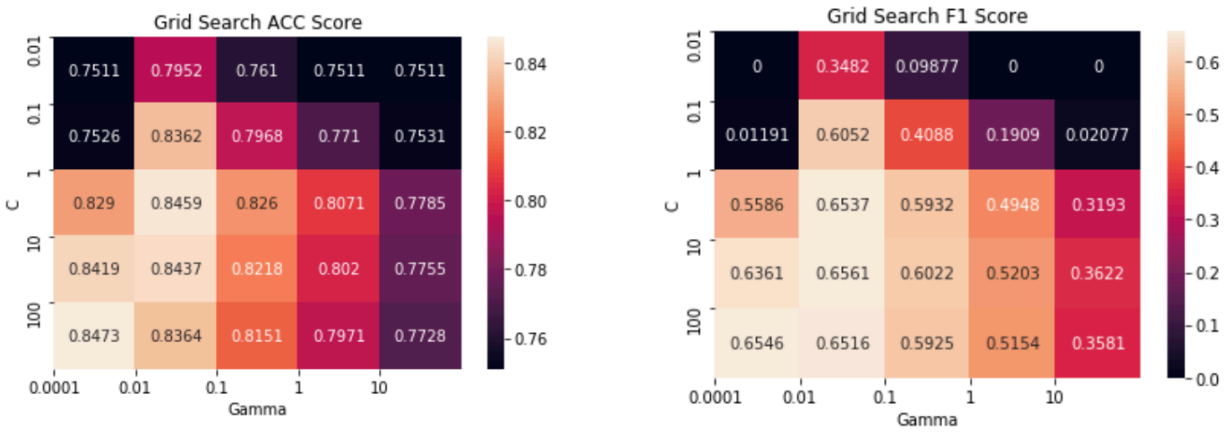


This heatmap generated using Seaborn [9] shows how accuracy and f1 score changes when n_estimators and max_features is changed. We see that increasing max_features improves the accuracy and f1 as are considering larger subsets of features and there are 70 different features. At small percentages of features such as 0.05 the model does not perform well as there is likely underfitting so the model cannot make predictions accurately. However, we do not want to consider too many features as this will lead to overfitting and decrease generalization decreasing performance on unseen data. In general, we see that increasing the number of estimators improves the performance as there are many different subsets of the data considered to make the many decorrelated trees, so we have less overfitting. However, increasing the number of trees to high will be more expensive to compute.

	param_max_features	param_n_estimators	mean_fit_time	mean_score_time	mean_test_f1	std_test_accuracy	std_test_f1
0	0.05	50	1.668664	0.158251	0.653986	0.001178	0.004200
1	0.05	75	2.304660	0.253157	0.655893	0.001645	0.005273
2	0.05	100	2.901955	0.357807	0.658988	0.001537	0.005883
3	0.05	200	6.260088	1.006630	0.656568	0.001625	0.003909
4	0.05	400	13.649576	2.160338	0.658407	0.001259	0.003987
5	0.05	600	19.952085	2.290052	0.656890	0.001340	0.003690
6	0.07	50	1.731761	0.207216	0.656756	0.002274	0.006921
7	0.07	75	3.072159	0.357240	0.660446	0.002195	0.004318
8	0.07	100	3.365852	0.322316	0.659060	0.001868	0.003809
9	0.07	200	6.253721	0.642416	0.661996	0.001155	0.003313
10	0.07	400	12.158315	1.241568	0.661610	0.001384	0.004547
11	0.07	600	18.046524	1.896616	0.665834	0.001416	0.004976
12	0.1	50	1.778942	0.158712	0.665384	0.001554	0.006075
13	0.1	75	2.582937	0.229525	0.666087	0.001926	0.004784
14	0.1	100	3.471913	0.303811	0.668701	0.001938	0.005656
15	0.1	200	7.016767	0.596033	0.666382	0.002180	0.006834
16	0.1	400	13.935662	1.206259	0.667386	0.001613	0.005397
17	0.1	600	20.926056	1.799473	0.667674	0.001689	0.005197
18	0.15	50	2.061141	0.158217	0.666110	0.003553	0.010153
19	0.15	75	2.995639	0.226950	0.668850	0.001264	0.004671
20	0.15	100	3.994917	0.297634	0.669184	0.002059	0.005760
21	0.15	200	8.011917	0.591891	0.668026	0.001829	0.006639
22	0.15	400	16.119283	1.219344	0.671274	0.002595	0.007734
23	0.15	600	24.593863	1.770221	0.670045	0.002856	0.008440
24	0.2	50	2.381300	0.158422	0.667460	0.002853	0.008443
25	0.2	75	3.666658	0.234706	0.668437	0.003031	0.006852
26	0.2	100	4.917095	0.304170	0.670341	0.002975	0.008201
27	0.2	200	9.749986	0.609690	0.672865	0.002525	0.007927
28	0.2	400	17.432354	0.922791	0.671060	0.002005	0.006305
29	0.2	600	19.290229	1.073502	0.671472	0.001993	0.006598

From this table we can see that increasing the number of estimators increases the fit and score time of the algorithm. As a result, we want to pick a number of estimators that is not too high. The number of features does not change the time of fit and score as much. Looking at our heat map and considering the impact of computation time we can then pick the **number of estimators as 200 and the number of max_features as 0.2** to produce optimal performance. This parameter gives an **accuracy score of 0.84653 and f1 score of 0.67287.**

Support Vector Machines Hyperparameter Tuning



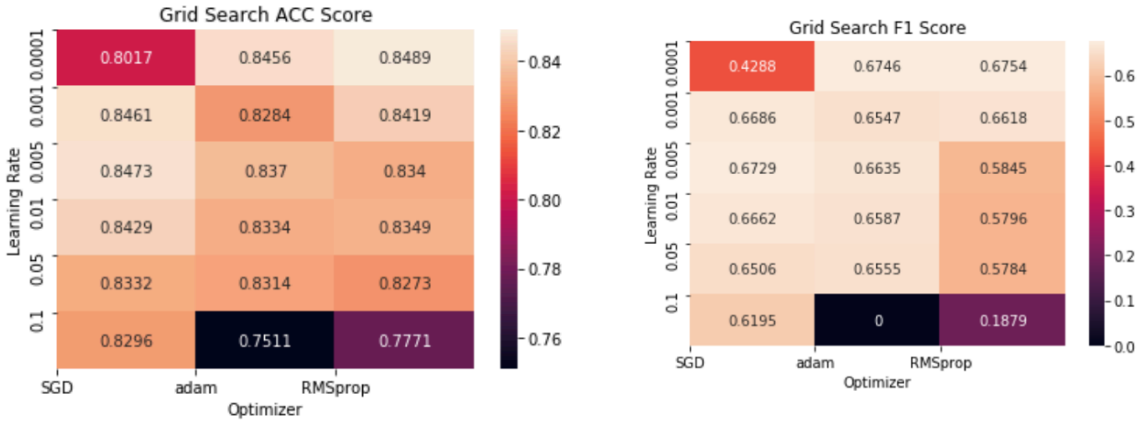
This heatmap shows the different scores for the gamma and C values tested. We see that increasing gamma decreases both accuracy and f1 score as the shape of the decision boundary becomes very closely fitted with the training data and thus, we end up with overfitting we use regardless of C value so when tested with unseen validation set the scores are low. If gamma is too low shape of the decision boundary is not complex enough and not fitted well so there is underfitting and we cannot get accurate results as the data is not captured effectively. However, if there is a large C value this helps improve the score for these low gamma values. For the C values, we see that increasing the value improves performance as we have a higher penalty for incorrect points. As a result, the margin is decreased and we fit the training model better, but we risk losing generalization. We see that if C value gets too high then in some cases we lose on accuracy as the model is more affected by noise in the training data, so we need to ensure we choose a value that is optimal.

	param_C	param_gamma	mean_fit_time	mean_score_time	std_test_accuracy	std_test_f1
0	0.01	0.0001	42.849624	14.369201	0.000047	0.000000
1	0.01	0.01	43.596223	14.308124	0.000141	0.002899
2	0.01	0.1	83.663728	20.327277	0.001922	0.018580
3	0.01	1	136.524395	25.855849	0.000047	0.000000
4	0.01	10	139.894881	28.311620	0.000047	0.000000
5	0.1	0.0001	44.226307	15.772761	0.000338	0.002326
6	0.1	0.01	38.594961	12.275706	0.001146	0.001415
7	0.1	0.1	147.856899	20.615727	0.000780	0.009021
8	0.1	1	248.822768	22.707074	0.000755	0.006551
9	0.1	10	185.651791	30.141216	0.000372	0.002941
10	1	0.0001	45.237655	14.902803	0.002305	0.003761
11	1	0.01	35.675571	10.692236	0.004190	0.012200
12	1	0.1	163.404909	17.232402	0.001770	0.004765
13	1	1	257.387302	21.705109	0.001362	0.007579
14	1	10	223.169053	25.671375	0.000893	0.004715
15	10	0.0001	33.774750	10.678527	0.002687	0.005244
16	10	0.01	38.791021	10.422880	0.005529	0.013953
17	10	0.1	206.263022	16.329393	0.002433	0.006466
18	10	1	245.686428	22.149351	0.002359	0.007359
19	10	10	286.303033	29.021807	0.001524	0.003485
20	100	0.0001	32.812120	10.681447	0.004282	0.010248
21	100	0.01	73.850449	11.786355	0.004539	0.009326
22	100	0.1	308.353724	19.194992	0.002489	0.004900
23	100	1	391.864033	18.394160	0.001817	0.007135
24	100	10	304.536635	17.750776	0.002411	0.004058

We can see from this table that as the gamma value gets higher the fit time increases greatly and score time increases so this makes the computation expensive as the decision boundary is more complex. The general trend for increasing C also causes the fit time to increase. Considering both the computational costs with training time and performance scores, the optimal hyperparameters are **0.01 gamma** and **1 for the regularization parameters C. This gives a f1 score of 0.6537 and accuracy score of 0.8459.** The highest f1 score was 0.6561 for C = 10 and gamma = 0.01. However, I calculated the t score as $\frac{\mu_1 - u_2}{\sqrt{\frac{s_1^2}{n} + \sqrt{\frac{s_2^2}{n}}}}$ which is $\frac{0.656081 - 0.653660}{\sqrt{\frac{0.013953^2}{3} + \frac{0.0122^2}{3}}} =$ 0.2262. The null hypothesis states there is not statistical difference between the two sample means. The critical value with a 95% confidence and n1 + n2 – 2 = 4 degrees of freedom as there are n samples is 2.776 with two tails. Since our t value lies between -2.776 and 2.776 we do not reject null

hypothesis so there is no statistical significance. Increasing C to 10 also increases the training time and slightly decreases accuracy, so we choose the C as 1.

Neural Network Hyperparameter Tuning



From this we can see that increasing the learning rate changes the scores and causes decreases in both F1 and accuracy score. A smaller learning rate allows the model to take smaller increments toward the optimal however it may take too long to converge, and we will need to increase the number of epochs and we can see poor results with some optimizers and the lowest learning rate of 0.0001. However, a large learning rate the optimal loss may have been overshot and it may have never converged leading to the poor results with the larger learning rates such as 0.1. Different optimizers produce better scores based on the learning rate that we use, and we see these changes in the heatmap.

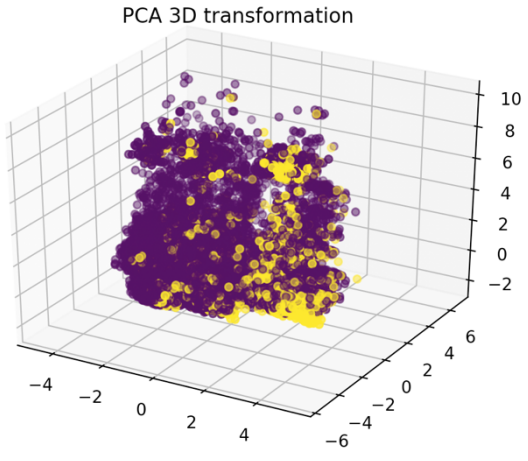
	param_optimizerS	param_learning_rate	mean_fit_time	mean_score_time	std_test_accuracy	std_test_f1
0	SGD	0.0001	196.923870	0.309599	0.022716	0.188546
1	adam	0.0001	209.067654	0.344351	0.001584	0.003960
2	RMSprop	0.0001	201.402509	0.395010	0.003546	0.008154
3	SGD	0.001	201.920363	0.444129	0.001431	0.004927
4	adam	0.001	219.745181	0.473226	0.005627	0.009808
5	RMSprop	0.001	214.244016	0.456135	0.005189	0.008233
6	SGD	0.005	211.044552	0.386210	0.003385	0.010350
7	adam	0.005	227.751129	0.406876	0.005932	0.008615
8	RMSprop	0.005	201.172616	0.464212	0.007398	0.063315
9	SGD	0.01	194.600365	0.415698	0.003533	0.016169
10	adam	0.01	208.716117	0.473467	0.004142	0.004976
11	RMSprop	0.01	193.942519	0.397049	0.012131	0.082755
12	SGD	0.05	183.729451	0.390364	0.002886	0.010698
13	adam	0.05	198.850801	0.434586	0.003354	0.044046
14	RMSprop	0.05	195.813444	0.423495	0.015492	0.098825
15	SGD	0.1	189.309617	0.359533	0.003497	0.027725
16	adam	0.1	164.132847	0.178479	0.003574	0.000000
17	RMSprop	0.1	157.493782	0.367847	0.037559	0.265729

The time taken to fit the model is highest for the adam optimizer. The SGD mostly has a quicker fit time when compared to RMSprop. As the learning rate increases the fitting time increases and then decreases when it gets even higher as we see the highest fit times are for learning rates of 0.005. The score time has variations and does not follow a specific trend. The two highest scores are for a learning rate of 0.0001 and the adam and RMSprop

optimizer. Since the RMSprop has higher F1 and accuracy scores and a quicker fit time, the chosen hyperparameters are **optimizer of RMSprop and learning rate of 0.0001.** This produces an **accuracy score of 0.8489 and f1 score of 0.6754**.

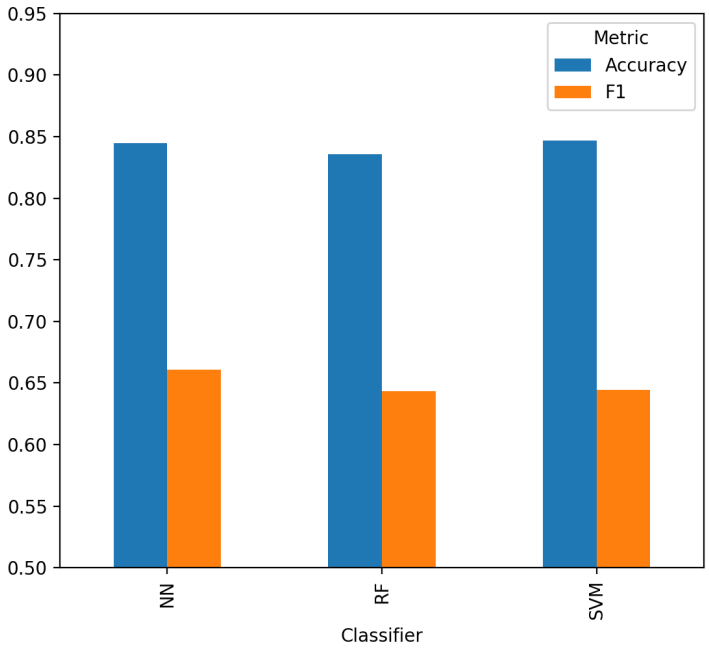
Nontriviality

The reason I believe my dataset has a non-trivial distribution is because we can see that we are considering many different features to make predictions based on them. The accuracy score is fairly high on the model (approximately 84%) but the f1 score is not as high (approximately 66%). This shows that when considering what classes are being predicted and precision of the model there is an impact on the performance. Shown in the figure is a projection of the data into 3 dimensions from 70. We can see that there are high areas of purple and yellow, especially purple class since this is the class with '<50k' which makes up approximately 75% of the dataset. However, there are many areas of the other yellow class '>=50k' within the other purple class and vice versa so there is no clear hyperplane that can be drawn to separate these classes as the two classes have lots of overlap. This indicates that the data is not highly linear separable.



Comparing Algorithm Performance

I used 30-fold cross validation and trained each classifier with the optimal hyperparameters selected after tuning using the complete testing data set. This means that the testing set is split into 30 subsets and one subset is used as the validation set to measure performance. This ensures that we were not just lucky with a split selected and our scores are more valid. I also used a 95% confidence interval and calculated the margin using the formula $\pm 1.96 * \frac{\sigma}{\sqrt{3}}$ to take into statistical significance between results when choosing the best classifier based on the Central Limit Theorem on the assumption of a Gaussian distribution. I measured performance using the mean accuracy and f1 score on the testing dataset which was unseen during the tuning process. Below, I have displayed bar graphs and a table representing the f1 and accuracy scores for all the three classifiers.



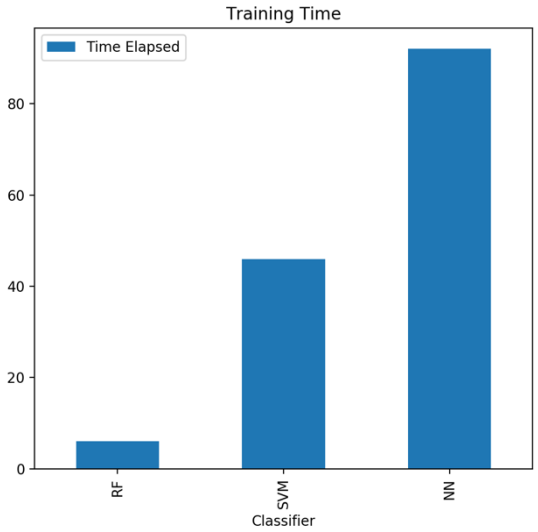
Classifier/Metric	F1	Accuracy
Random Forest	0.65425295 $\pm$0.01114	0.84050465$\pm$0.004493

SVM	0.6446869 ± 0.0124894	0.8467463 ± 0.0048798
Neural Network	0.6581396 ± 0.0101530	0.8460823 ±0.0048485

We can see from these results that the different classifiers produce similar results. The SVM produces the lowest F1 score compared to the other classifiers. However, the difference is not statistically significant as the confidence intervals overlap. The highest f1 score is produced by the neural network. In terms of accuracy, the random forest produces the lowest accuracy followed by the neural network and then the SVM. However, these differences are not statistically significant either because of overlapping confidence intervals. Since it produces the highest f1 score and a good accuracy score, the neural network would be the best classifier for this dataset. However, we should consider other factors such as training time to pick the best classifier for this dataset which is done in the next section since performance using the classifiers is similar.

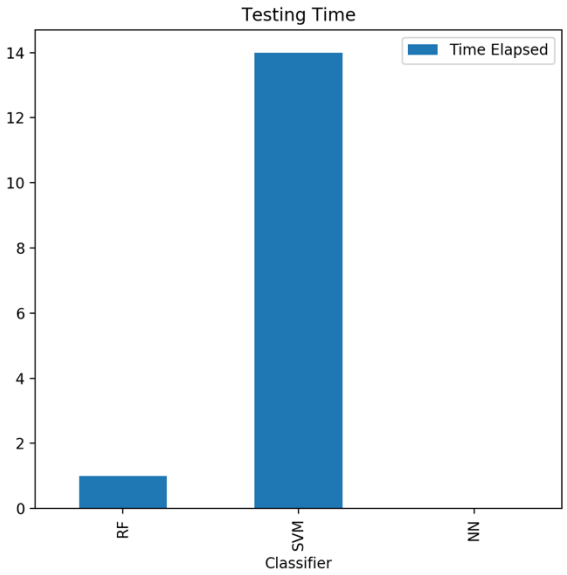
Conclusion

I have plotted the time taken to fit models to each of the different classifiers. It is important to take this into consideration when choosing a classifier as this would be more expensive to compute.



Random Forests have the shortest training time of 6 seconds despite the large size of the dataset and 200 different trees selected. Following this is the SVM classifier with a training time of 46 seconds. This is slower as it is a quadratic programming problem and we have both a large number of features and instances [12]. The highest time taken is by the neural network as it takes 92 seconds to train. This is because of the number of epochs selected as 150 which requires iterating through the set consuming time for the gradient descent to converge, and we have 4 layers also affecting the time taken.

A similar figure was created but in terms of the time taken to evaluate the testing set and predict outputs.



From this graph we see that predicting values for the testing set for the random forest and neural network were both very fast as they took under 1 and 0 seconds respectively. However, the SVM took 14 seconds to make predictions. This could be because there are a large number of support vectors in the model based on the regularization parameter set. However, increasing the parameter could affect the performance so this is a trade-off between the performance and computation time.

Based on the performance and both the training and testing time, the algorithm that should be used is the random forest. This classifier has the quickest training time, a very quick testing time and produces results that are very close to the results generated by the SVM and NN so the difference is not statistically significant. This algorithm works well on this large dataset with many features as we use bagging and work with subsets of the data making it efficient in computation. Because we are using census data that will be very large in size, we want a classifier that will be efficient in computation. Since we consider many different decision trees, we can also reduce variance by considering the mode class predicted. The hyperparameters I would use for the random forest is number of estimators as 200 and max features as 0.2 and defaults for the remaining hyperparameters.

Acknowledgements:

[1] Child Poverty in America 2017:National Analsys. https://www.childrensdefense.org/wp-content/uploads/2018/09/Child-Poverty-in-America-2017-National-Fact-Sheet.pdf

[2] https://archive.ics.uci.edu/ml/datasets/Census+Income

[3] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

[4] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.score

[5] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A Practical Guide to Support Vector Classification. May 19, 2016. https://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf

[6] Olier Knocklein. Classification using Neural Networks.

https://towardsdatascience.com/classification-using-neural-networks-b8e98f3a904f

[7] https://keras.io

[8] https://scikit-learn.org/0.16/modules/generated/sklearn.grid_search.GridSearchCV.html

[9] https://seaborn.pydata.org/generated/seaborn.heatmap.html

[10] https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/

[11] https://stackoverflow.com/questions/47796264/function-to-create-grouped-bar-plot

[12]https://scikit-learn.org/stable/modules/svm.html#complexity