Predicting Data Scientist Salaries

Katherine Keiko Dyo
Department of Natural Sciences and Mathematics
The University of Texas at Dallas
Richardson TX, USA
kkd150130@utdallas.edu, Katdyo@gmail.com

James Boyer
Department of Natural Sciences and Mathematics
The University of Texas at Dallas
Richardson TX, USA
jsb170130@utdallas.edu, skylerboyer@gmail.com

Kirti Chhatwal
Department of Electrical and Computer Engineering
The University of Texas at Dallas
Richardson, TX, USA
kxc20024@utdallas.edu, kirtichhatwal2@gmail.com

Manjula Mahesh K Ranpati Dewage
Department of Natural Sciences and Mathematics
TheUniversity of Texas at Dallas
Richardson, TX, USA
mkr200000@utdallas.edu, manjulamahesh274@gmail.com

***Abstract*—This project aims to implement supervised and unsupervised learning algorithms relevant to data science salaries.**

***Keywords*—Big Data, Machine Learning, Data Science, PySpark, MapReduce, DataBricks**

I. Introduction

The project utilized the Global AI/ML salaries dataset containing information about salaries for data science jobs from ai-jobs.net. The dataset contains 6303 observations and 11 variables (year, level of experience, employment type, job title, salary, salary currency, salary in USD, employee residence, remote ratio, company location, company size). Two supervised learning algorithms were implemented: linear regression and decision tree. Additionally, two unsupervised algorithms were applied: principal component analysis (PCA) and k-means clustering (using our implementation of MapReduce and MLlib's implementation).

II. Data Preparation

A. *Exploratory Data Analysis*

Our first step was to investigate the distribution of our target feature (*salary_in_usd)* and predictive features. We observed that the salaries seemed to follow a normal distribution.

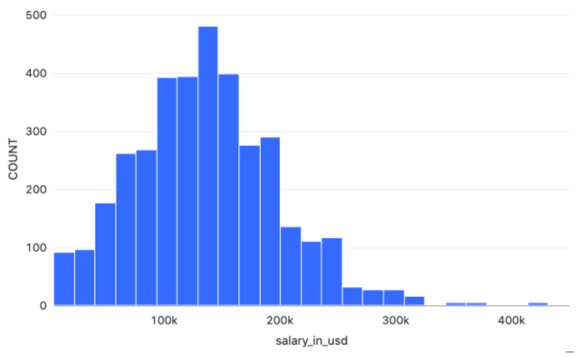

Figure 1: Histogram of *salary_in_usd*

Our next observation was that two of our features contained a large quantity of values. First, *job_title* had 83 different values with many of them being similar but phrased slightly differently. We observed that different terms were associated with varying salaries such as analyst being associated with lower salaries than manager.

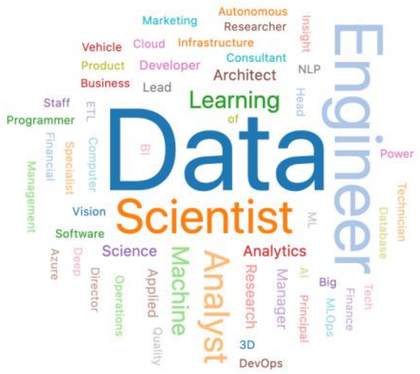

Figure 2: Word cloud of most common terms in *job_title*

Second, our *employee_residence* feature had 80 different values with over 80% of employees in the dataset living in the United States.

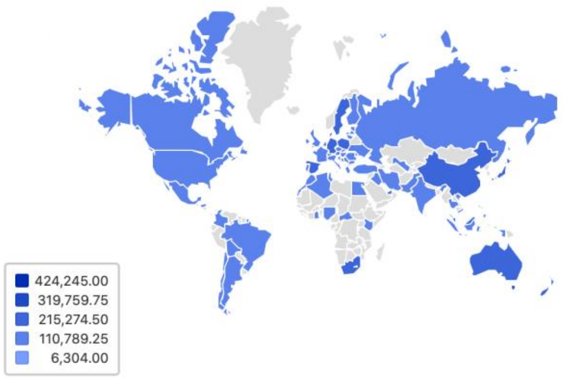

Figure 3: Average Salary by Country

B. *Feature Engineering Pipeline*

We made the following modifications to our dataset to make it more amenable to analysis.

First, we applied a logarithmic transformation to the *salary_in_usd* feature because the range of salaries spanned several orders of magnitude and we wanted the range of values to be less extreme.

Second, we numerically encoded *remote_ratio, experience_level,* and *company_size* using the natural ordering of those values. For example, we ordered lowest to higheset experience levels on a scale of 0 to 3.

Third, we replaced the *employment_type* feature with a binary feature of whether or not the employee was full time or not.

Fourth, we mapped the *job_title* feature, which originally contained 83 different values, to one of five different categories: engineer, manager, analyst, scientist and developer. We then numerically encoded these labels.

Fifth, we replaced the *employee_residence* feature that specified what country the employee lived in with a feature of what continent the employee lived in. We then one hot encoded this feature. We also dropped *company_location* as a feature because it was almost identical to *employee_residence* and the two disagreed less than 50 times.

Sixth and finally, we normalized each feature independently to have zero mean and unit variance.

We converted all these stages into a single process using MLlib's pipeline API to automate this entire process.

III. Supervised Learning Experiments

We began our investigation by testing to what extent we could predict the salaries of data scientists within our dataset. We settled on two key approaches that were also both implemented in Spark's MLlib library: Linear Regression (and regularization) and Decision Trees.

A. *Linear Regression*

The dataset was split randomly and 80% was used for training linear models while 20% was used for testing the model. The features pre-processed features vector was set as the "featuresCol" and the target variable was the log transformed salary_in_usd. The linear models utilized MLLib linear regression with the regParam value being set to zero for ordinary least squares and 0.1 for both ridge and LASSO methods. For ridge regression, the elasticNetParam value was set to zero while LASSO used an elasticNetParam value of 1.

Ordinary least squares linear regression was applied yielding an RSME of 0.40 and 0.42 on the training and testing data respectively.

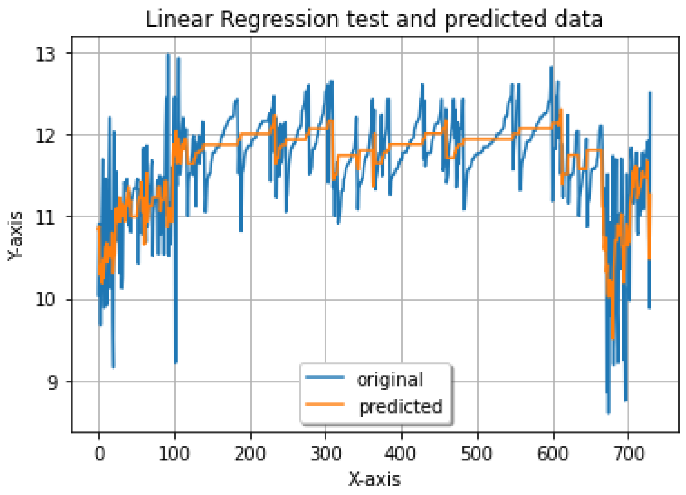

Figure 4: Performance of Linear Regression

Using Ridge regression resulted in RSMEs of 0.41 and 0.42 for training and test data. LASSO regression performed the worst with an RSME of 0.44 on the training data and 0.46 on the testing data. The LASSO model does not perform well predicting the original values as seen in the graph below.

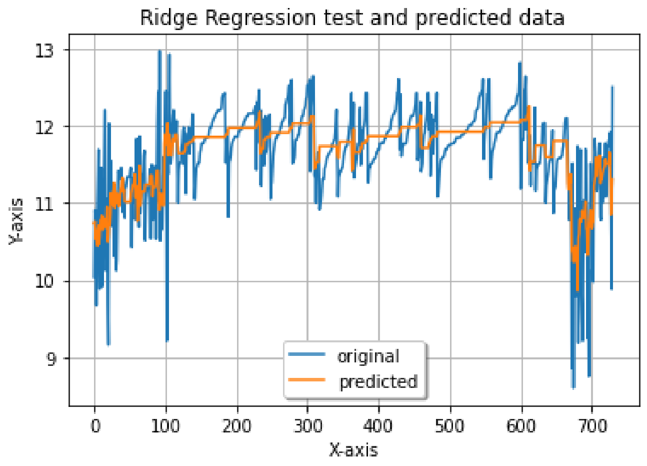

Figure 5: Performance of Linear Regression with Ridge Regression

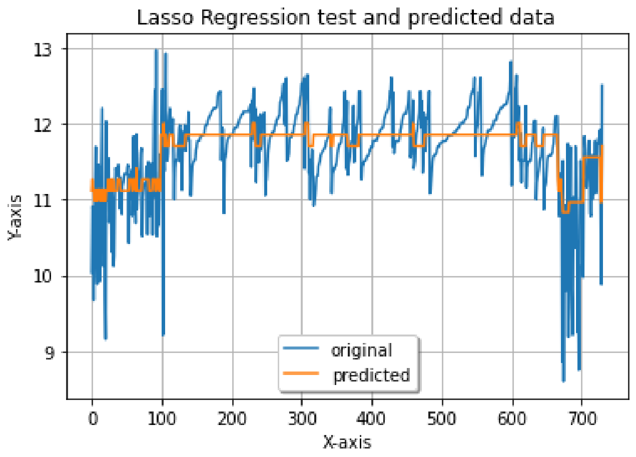

Figure 6: Performance of Linear Regression with Lasso Regression

Both the OLS and Ridge models had an R squared value of 0.51 on the testing data while the LASSO model had an R squared value of 0.42. Based on the results, the OLS model seems to be the best because applying the ridge penalty does not result in a significant improvement, and visually the two models resulted in very similar predicted salary values.

B. Decision Tree

In machine learning, a decision tree is a model used to make decisions by representing decisions and their results in a tree structure. The final decision is represented by leaves, and it contains internal decision nodes. In the context of regression problems, decision trees can be used to estimate values based on input parameters. The leaf nodes of the decision tree contain the predicted value. Here, we use the PySpark machine learning library to build a decision tree regression model to predict an employee's salary based on various inputs.

To begin with, we divide our dataset into training and test sets using the "randomSplit()" method. Next, we create a DecisionTreeRegressor object, setting the "features" column to "scaledFeatures" and the "label" column to "salary_in_usd_log". To optimize the model's performance, we construct a grid of hyperparameters using the "ParamGridBuilder()" method, varying the "maxDepth", "minInstancesPerNode", and "maxBins" values. We then employ a CrossValidator object to perform a 5-fold cross-validation to determine the best set of hyperparameters for our model.

For the DecisionTreeRegressor, we tested the following hyperparameters:
maxDepth: 2, 5, 10 minInstancesPerNode: 1, 10, 20
maxBins: 10, 20, 30

After tuning the hyperparameters using 5-fold cross-validation, we found that the optimal set of parameters for this dataset were:

- maxDepth: 10
- minInstancesPerNode: 20
- maxBins: 30

Once the model has been trained using the training data, we make predictions on the test data using the "transform ()" method. We then evaluate the model's performance by calculating the root mean squared error (RMSE) and R-squared (R2) metrics using the RegressionEvaluator object. The RMSE indicates the average difference between the predicted and actual salaries, while the R2 value represents the proportion of the variance in the data that the model is able to account for.

In order to visually represent the performance of the model, we can create a scatter plot comparing the predicted salaries to the actual salaries.

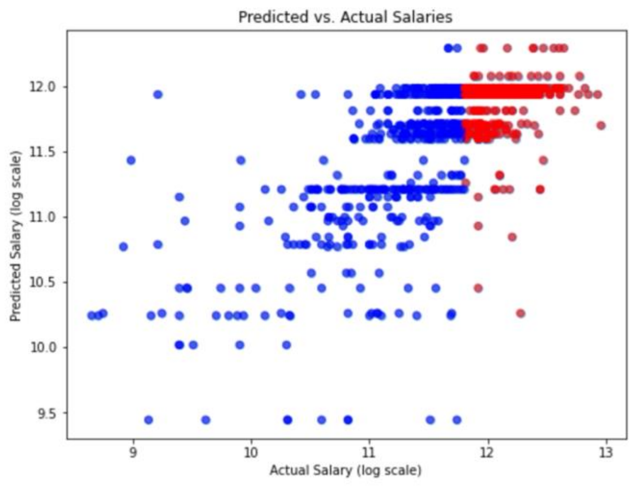

Figure 7: Performance of Decision Tree (Predicted vs. Actual Salaries)

Here, the horizontal axis of the graph indicates the real salaries (measured in log scale), whereas the vertical axis shows the projected salaries (also in log scale). Each point in the scatter plot corresponds to an employee's salary, and the color of each point indicates whether the actual salary is above or below the median actual salary.

We can explore additional hyperparameters such as maxMemoryInMB, which sets the maximum memory allowed for collecting statistics, and minInfoGain, which specifies the minimum information gain required for a split to occur. Additionally, we can explore other regression algorithms like Random Forest or Gradient Boosted Trees, which might provide better performance than the DecisionTreeRegressor for this dataset.

IV. Unsupervised Learning Experiments

We continued our investigation by exploring two unsupervised approaches to identifying patterns in our data: Principal Components Analysis (PCA) and K-Means. We used PCA to determine the extent we could reduce the dimensionality of our data. We used K-Means to evaluate whether our data exhibited natural clusters reflecting categories of data scientists. For K-Means we also decided to implement our own version using MapReduce and compared it to MLlib's implementation.

A. *Principal Component Analysis*

Our first step in investigating our data was to analyze to what extent we could reduce the dimensionality of our data. We performed Principal Component Analysis and determined that each of the 11 features we had extracted explained roughly 10% of the variation in our data.

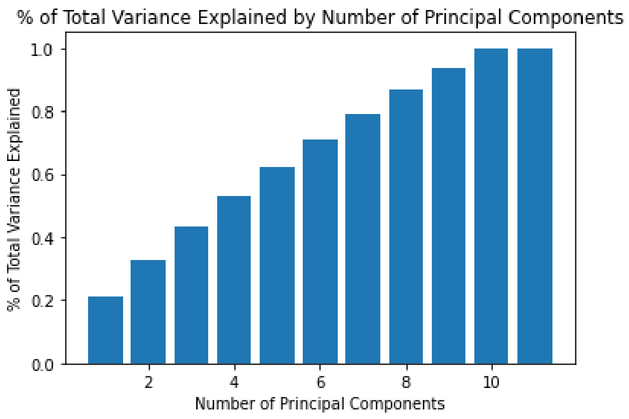

Figure 8: The total amount of variation explained by the number of principal components.

Our first conclusion is that our features cannot be easily reduced to a lower dimension without losing crucial information about the data. Similarly, since the amount of variation explained by each of the principal components is roughly equal, we also conclude that our features are independent of each other which helps us avoid any of the problems associated with collinearity.

B. *K-Means using MapReduce*

As part of this project's requirements, we decided to implement the K-Means algorithm using MapReduce. To evaluate the performance of a given set of centroids, we calculated the mean squared sum of distances between every point and its respective centroid across all clusters again using MapReduce. To determine the optimal number of clusters, we plot this mean squared sum against the number of clusters and determine where the elbow point falls. This metric, which we abbreviate as MSS, is specified below at (*) with the following variables: n is the number of data points, k is the number of clusters, C_i is the i^{th}cluster, c_i is the center of i^{th} cluster, and $x_j$represents data points in C_i.

$$MSS = \frac{1}{n}\sum_{i=1}^{k}\sum_{x_j \in C_i}\left(x_j - c_i\right)^2 \quad --------(*)$$

With our metric defined, we implemented K-Means as follows:

Algorithm 1:

1. Start with an initial cluster assignment C by choosing k random points from our dataset to be our centroids.
2. Iterate for a pre-determined number of iterations (maxIter) or until cluster assignments stop changing with respect to a tolerance threshold (we simply a sufficiently large maxIter for the purposes of this project):
    a. For the current $\{c_i\}_{i=1}^{k}$, we minimize MSS using **map** by assigning each observation to the closest centroid.
    b. For the current C, we minimize MSS with respect to $\{c_i\}_{i=1}^{k}$ by finding the average point in each cluster using **reduceByKey.**

Remark: Using MSS is equivalent to SS , which is $n.MSS$. For efficiency purpose we used SS in the code.

Our K-Means algorithm accepts five inputs: k (integer) - the number of clusters, df (Dataframe) a dataframe with prepared column of vectors to be used for clustering, colName (string) - name of the feature column to cluster, seed (integer) - number of random seed, and maxIter (integer) - maximum number of iterations. Our algorithm outputs the cluster assignments and the final centroids.

To find the optimal number of clusters, we computed the average distance (MMS) given by equation (*) for a range of number of clusters. We wrote a function called averageDistance to compute MSS as a function of assignments and centroids. We then plot MSS vs. number of clusters to determine the elbow point and hence our ideal number of clusters.

We tried computing MSS for k ranging from 1 to 20. We observed that the two following graphs are not strictly decreasing as they show some random increments and decrements. This is due to the random seeds. In the K-Means algorithm in standard libraries such as Spark MLlib and Scikit learn fit the model for several seeds and then give the best model with respect to built-in metric. This is a type of improvement to our method for future work.

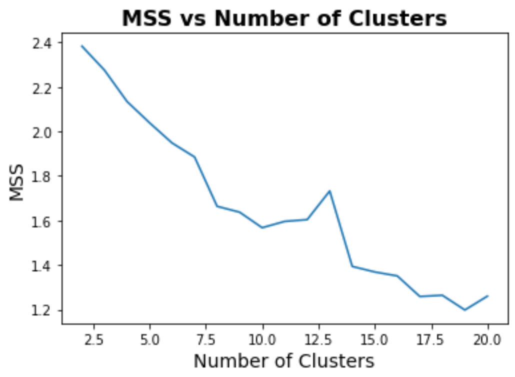

Figure 9: The graph of MSS vs number of clusters considering all features. The elbow point can be observed at k=16.

We next decided to try clustering using the two most significant principal components. We observed a much cleaner elbow point indicating that the data projected onto the first principal components was much more amenable to clustering.

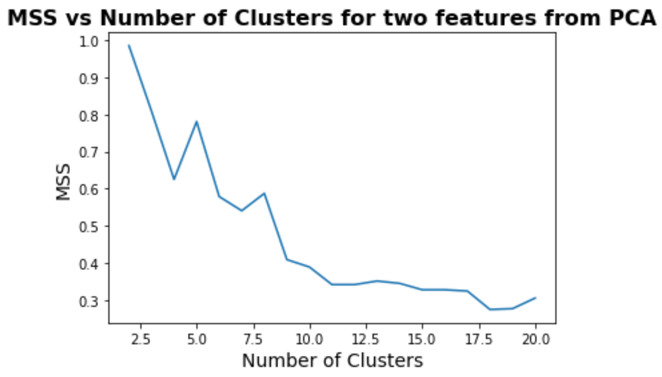

Figure 10: The graph of MSS vs. number of clusters considering for two features obtained from PCA. The elbow point can be observed at k=9.

After a visual investigation of the plot of the first two principal components, we visually identified three major clusters. We decided to test the performance of our K-Means algorithm when k=3.

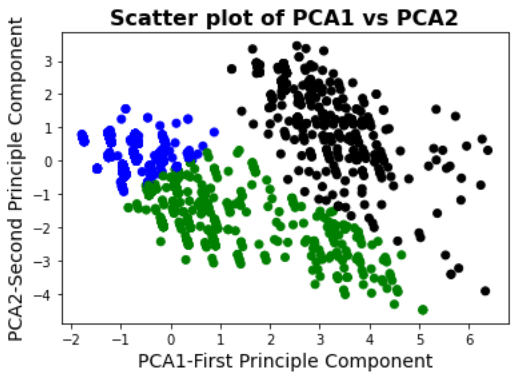

Figure 11: The scatter plot for two features obtained from PCA with **number of clusters k=3** using MapReduce version of the K-Means algorithm.

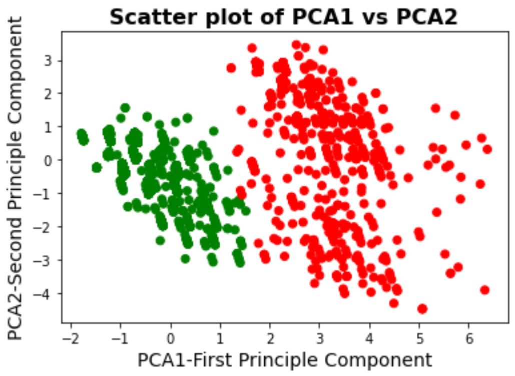

Figure 12: The scatter plot for two features obtained from PCA with **2 clusters** using MapReduce version of the K-Means algorithm. We include this to contrast what happens to clustering around optimal number of clusters.

C. *K-Means using MLlib*

We then experimented with MLlib's K-Means implementation using the visually determined optimal number of clusters k=3. We found that MLlib's version performed better than ours. We attribute this to MLlib's version testing a variety of random seeds and then selecting the optimal performance and then choosing the best one.

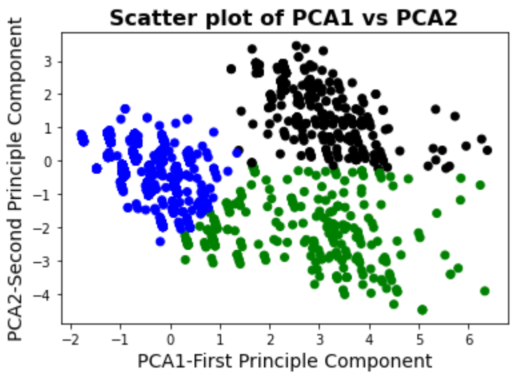

Figure 13: The scatter plot for two features obtained from PCA with **3 clusters** using K-Means algorithm in Spark MLlib.

D. *Comparing K-Means Algorithms*

There are three ways the K-Means algorithms can be compared.

First, the mode of implementation. Our implementation was primarily a functional approach which we believe could be easily extended into a class-based structure like MLlib's version. We could also convert our algorithm into a Transformer/Estimator so that it could be included in an

MLlib pipeline and automatically identify centroids and label datapoints.

Second, the performance of the algorithm. We used MSS whereas MLlib's implementation includes the Silhouette metric. We could also improve the performance by setting the stopping criteria to an epsilon value that when the centroids no longer change by a value greater than epsilon, we terminate the algorithm. By using maxIter=10 for most of our experiments, there might have been a hit on the overall performance of our clustering.

V. Conclusion

This project provided three main takeaways. First, we learned about the value of automating the workflow using the pipeline API. Even though we had four people working on this project, our capacity to share code and work together was enhanced by the fact that each of us only had to be concerned with the pipeline object and the specifics of individual steps. Second, in Supervised Learning, the use of the decision tree algorithm proved to be a suitable choice for accurately predicting salaries in the field of data science by capturing the relationships between various variables. Decision trees are also easy to interpret, making it easier to understand the factors that influence salaries. Third, our Unsupervised Learning experiments helped us develop a stronger intuition behind the strengths and weakness of K-Means. We found that K-Means excels in its simplicity and efficiency but is prone to fall into suboptimal clusters.

References

1. James, G., Witten, D., Hastie, T., & Tibshirani, R. (2022). *An introduction to statistical learning: With applications in R*. Springer.
2. *Machine Learning Library (mllib) guide*. MLlib: Main Guide - Spark 3.4.0 Documentation. (n.d.). Retrieved April 29, 2023, from https://spark.apache.org/docs/latest/ml-guide.html
3. Choudhary, P. (2023). Statistical and Machine Learning [Class handout]. The University of Texas at Dallas, Richardson, STAT 6340.
4. Yorkinov, O. (2021, May 20). *MLLib linear regression example with pyspark*. MLLib Linear Regression Example with PySpark. Retrieved April 29, 2023, from https://www.datatechnotes.com/2021/05/mllib-linear-regression-example-with.html