Loan Acceptance Predictor Report

Anson Yamvinij, Ishaan Sathaye, Michael Montemurno

Introduction:

We found this dataset on Kaggle and wanted to learn how different machine-learning algorithms would perform on the same problem. We put ourselves in the shoes of a data science team working for the bank, working to develop the best predictive model. This model will predict if a loan applicant will struggle to pay their loan.

Methodology:

In this project, we will be doing some pre-processing on the CSV data and converting it to RDDs. To perform some exploratory data analysis, we will be taking a look at the data and finding interesting graphs, outliers, and relying on domain knowledge to observe which model would be better to fit the data. After that, we will be implementing from scratch 3 algorithms which are logistic regression, k-nearest classifier, and naive bayes algorithm. For each of the algorithms we are splitting the data on a 80-20 split with a training and validation set. For our metrics, we will be calculating the precision, recall, accuracy, and f1 score to evaluate the model ultimately on the validation set.

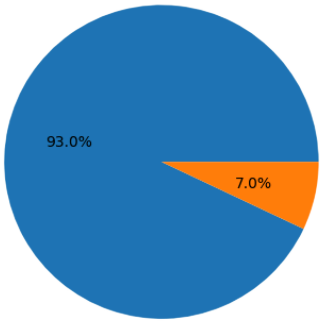
Data Processing and Exploration:

We obtained the dataset from Kaggle as we mentioned above and it contained multiple files that can be joined, however the applications.csv contained the most important features to be considered. This data was not clean so we had to pre-process the data in terms of dropping missing applications, converting units, and dropping features. We did this using python in a jupyter notebook where we converted the time to appropriate units and also relied on domain knowledge to keep the important features.

To format this to RDD, we used SparkContext and used pattern matching to handle the various cases in the data and mapped them to an Array. In the case matching we converted all the categorical variables in a one-hot encoded manner. Everything was converted into a double for a simpler dataset. For any of the rows that remained which did not have entries we added a special case which defaulted in 0 for that feature. This enabled

```
case class LoanApplicant(
  SK_ID_CURR: Double,
  TARGET: Double,
  NAME_CONTRACT_TYPE: Double,
  FLAG_OWN_CAR: Double,
  FLAG_OWN_REALTY: Double,
  AMT_INCOME_TOTAL: Double,
  AMT_CREDIT: Double,
  AMT_ANNUITY: Double,
  AMT_GOODS_PRICE: Double,
  REGION_POPULATION_RELATIVE: Double,
  CNT_FAM_MEMBERS: Double,
  REGION_RATING_CLIENT: Double,
  EXT_SOURCE_2: Double,
  EXT_SOURCE_3: Double,
  YEARS_BEGINEXPLUATATION_AVG: Double,
  OBS_60_CNT_SOCIAL_CIRCLE: Double,
  DEF_60_CNT_SOCIAL_CIRCLE: Double,
  AMT_REQ_CREDIT_BUREAU_YEAR: Double,
  AGE: Double,
  YEARS_EMPLOYED: Double,
  YEARS_REGISTERED: Double,
  YEARS_ID_PUBLISH: Double
)
```

us to handle all the invalid or errored data. After that we saved this final dataset as a text file to the output folder after running App.scala, from where each of our algorithms will read the dataset.



Majority are all other cases of loan statuses vs Minority of loan repayment difficulties

Models Implemented:

Logistic Regression

We explored using a logistic regression model first which is a statistical model that is widely used for binary classification problems. This model predicts probabilities and maps them to binary outcomes using the sigmoid function.

The dataset contained multiple features and a binary variable which is the loan defaulting variable. The model was fitting for predicting this for loan applications since it directly models the probability of the occurrence of the event. However, there are several challenges that came with fitting this model such as class imbalance, overfitting on the training data, and hyperparameter tuning.

In this implementation of logistic regression, first all the features were scaled to have a mean of 0 and standard deviation of 1, which ensured that features contributed equally to the gradient calculations during optimization. Scaling was performed on all features after assembling them into a feature vector. Vector Assembler was used to combine individual features into a single feature vector, which simplified computations in Spark's distributed environment:

```scala
// Define features and assemble them
    val featureColumns = columnNames.drop(2)
    val assembler = new VectorAssembler()
      .setInputCols(featureColumns.toArray)
      .setOutputCol("features")
```

Class imbalance was handled by creating class weights and then those weights were incorporated into the gradient descent algorithm to penalize the minority class less:

$$\text{Weight} = \frac{\text{Total Samples}}{\text{Number of Classes} \times \text{Class Frequency}}$$

To help reduce overfitting, in the gradient descent a L2 regularization term was introduced which penalized larger weights:

```scala
// Train the model using gradient descent
    for (_ <- 1 to maxIterations) {
      val gradients = trainingRDD.map { case (features, label) =>
        val linearCombination = features.zip(weights).map {
            case (f, w) => f * w }.sum
        val prediction = sigmoid(linearCombination)
        val error = prediction - label
        val weight = classWeights(label)
        features.map(_ * error * weight)
      }.reduce((g1, g2) => g1.zip(g2).map { case (x, y) => x + y })

      // Update weights
        val regularization = 0.315
        weights = weights.zip(gradients).map { case (w, g) => w -
            learningRate * g + regularization * w }
    }
```

When evaluating this model, we used the validation set with metrics such as accuracy, precision, recall, and f1-score. The accuracy this model achieved was 75%, a precision of 0.11, a recall of 0.29, and a f1 score of 0.16. This model struggled with detecting the minority class, as reflected in the low precision and recall, and specifically in the validation set 48,000 were classified as 0 while 13,000 were classified as 1 (difficulties). This was expected since although logistic regression is best when interpretability and simplicity are important, it assumes linearity, which can limit performance on complex datasets such as this. The other 2 models seem to be a better fit for this dataset as they are a good baseline for this categorical dataset as well as being flexible.

K-Nearest Classifier

The next model we explored was a K-Nearest Neighbors classifier, which is a model used for a wide variety of classification problems. Unlike the other two models, there isn't a distinct

"training" phase, as once you have the training data and the model constructed you immediately are able to start predicting points.

All of the data was standardized with Z-Score standardization, and a 80-10-10 split on the data was used for training, validation, and testing respectively.

The prediction method took in an RDD of the training data, the testing data, and how many nearest neighbors should be considered. The method then returns an RDD with the index of each datapoint and whether or not it predicts their application will be accepted.

```
def predict(train_data : RDD[(Double, (Double, Double, Double, Double, Double))],
            test_data : RDD[(Double, (Double, Double, Double, Double, Double))],
            k : Int) :  RDD[(Double, Double)]  = {
    // print(k + "\r")
    val revised_data = test_data
                        // do a cartesian product over all of the data
                        .cartesian(train_data)
                        // group by the key of the testing data
                        .groupBy(x => x._1._1)
                        // find the distance for each pair
                        .map(x => (x._1, x._2.map(pair => euclideanDistance(pair._1, pair._2))))
    // output an rdd with each prediction
    revised_data.map(x => (x._1, x._2.toList.sorted.take(k).fold(0.0)((total, n) => total + n )))
                .map(x => (x._1, if(x._2 / k.toDouble < threshold) 0.0 else 1.0 ))
}
```

The basic process done for this method is to pair every point in the test set with every point in the training data, and then to group by each point in the test set. The distance between each point is then calculated, ranked, and then averaged out. If the average rounds to above the threshold value, which I tuned to 0.55, then the value was marked as being accepted.

This model can suffer from something called "the curse of dimensionality", so to reduce that I limited each featureset I tested to four features. A few different feature sets were tested, testing different combinations of categorical and quantitative features. The five featuresets tested were:

1. Total Income, type of loan, age of applicant, and amount of credit
2. Total income, whether they own a car, a count of family members, and their amount of annuity
3. Amount of income, years employed, age, amount of credit
4. Years registered, years employed, age, and amount of annuity
5. Amount of total income, years employed, years registered, and the region rating of the client

All different feature sets were compared against each other with a constant k.

```
println(kHelpers.calcMetrics(kHelpers.makeConfusionMatrix(kHelpers.predict(train_data_1, val_data_1, k), val_data_1)))
println(kHelpers.calcMetrics(kHelpers.makeConfusionMatrix(kHelpers.predict(train_data_2, val_data_2, k), val_data_2)))
println(kHelpers.calcMetrics(kHelpers.makeConfusionMatrix(kHelpers.predict(train_data_3, val_data_3, k), val_data_3)))
println(kHelpers.calcMetrics(kHelpers.makeConfusionMatrix(kHelpers.predict(train_data_4, val_data_4, k), val_data_4)))
println(kHelpers.calcMetrics(kHelpers.makeConfusionMatrix(kHelpers.predict(train_data_5, val_data_5, k), val_data_5)))
```

(This is how you are able to get the metrics from my model, by predicting on the validation set with the training data, and passing that to the method that makes a confusion matrix, and then passing that to the method that calculates metrics from the confusion matrix)
Multiple different values of K were also tested, with the best one being around 10 for the portion of the dataset. The best feature set ended up being number 2, with the features total income, whether they own a car, a count of family members, and amount of annuity coming out on top.

The output of testing and refining my model was featureset number 2 with 10 nearest neighbors, which outputted, from left to right, accuracy, precision, recall, and f1 score.
Accuracy: 0.8985, Precision: 0.03731, Recall: 0.06849,
and F1 Score: 0.02415)

These metrics are not very good, as similarly to the Logistic Regression model the minority class was not identified well, as seen by the low precision and recall. This is likely because K Nearest Neighbors struggles a lot with unbalanced datasets and large datasets, which would very aptly describe this dataset. Ironically, my attempt to avoid the curse of dimensionality may have made me not be able to predict as well, as more features may have been able to distinguish the minority class better. Another struggle I had was testing on a smaller sample of the dataset, and my amount of neighbors and featureset not translating up to the larger version. The high accuracy was most likely achieved by the model simply labeling almost every point as a 1.0, with very few being able to reach 0.0, as there was likely not enough 0.0 points to influence a trend.

As a machine learning specialist advising the bank, I would say look elsewhere for a model rather than using K-nearest neighbors, as their larger datasets and diverse unbalanced datasets don't suit it well.

Naive Bayes Classifier

Unlike the other two models, the Naive Bayes classifier is a generative learning algorithm. This means that it does not learn what features are the most significant. This model is based on Bayes' theorem and assumes that all the features are independent and follow a normal distribution. Bayes' Theorem is $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$ which finds the probability of an event occurring given the probability of another

event that occurred already. This formula gets much more complicated with more features but it is still relatively simple and quick to compute.

The first step was standardizing the data so all features would be weighted equally. Then I used an 80/20 train/test split to simulate how the model would perform on unseen data. After that, I could begin implementing Naive Bayes.

Three main components were needed to reach the outcome of a predicted class. First, I needed the mean and variance of every feature.

```
val featureStats = features.groupBy(_._2).map { case (featureIndex, featureValues) =>
  val values = featureValues.map(_._1)
  val mean = values.sum / values.size
  val variance = values.map(v => pow(v - mean, 2)).sum / values.size
  (featureIndex, (mean, variance))
}
(targetClass, featureStats)
```

Then I needed to calculate the probability of being in each class.

```
val classProbs = classCounts.map { case (targetClass, count) =>
  (targetClass, count.toDouble / totalCount)
}
```

Once I had these numbers, I used log-likelihood to calculate the probability of the applicant being in each class.

```
val logLikelihoods = model.classProbs.keys.map { targetClass =>
  val classProb = log(model.classProbs(targetClass))
  val featureLikelihoods = featureValues.zipWithIndex.map { case (featureValue, featureIndex) =>
    val (mean, variance) = model.featureProbs(targetClass)(featureIndex)
    val likelihood = (-0.5 * log(2 * Pi * variance)) - (0.5 * pow(featureValue - mean, 2) / variance)
    likelihood
  }
  val logLikelihood = classProb + featureLikelihoods.sum
  (targetClass, logLikelihood)
}

val predictedClass = logLikelihoods.maxBy(_._2)._1
predictedClass
```

Finally, the class with the highest probability gets returned.

Confusion Matrix:

	Predicted 1	Predicted 0
Actually 1	1148	3278
Actually 0	3822	52275

Overall, this model had decent results with a majority of the applicants being correctly classified. More metrics will be displayed below.

Results and Discussion:

Model	Accuracy	Precision	Recall	F1-Score
Logistic Regression	0.75	0.11	0.29	0.16
KNN	0.90	0.037	0.068	0.024
Naive Bayes	0.876	0.231	0.235	0.233

We chose to focus on these metrics because we want to make sure our model is correct and not just approving every single applicant. As a bank, we do not want to lose money on defaulted loans. Accuracy measures the proportion of correct predictions compared to the total number of predictions. Precision measures how many of the retrieved elements are relevant. Recall measures how many relevant elements are retrieved. The F1 score is the harmonic mean of precision and recall so it combined them into a single number.

K-Nearest Neighbors was our most accurate model with an accuracy of 0.90. Naive Bayes was not that far behind with an accuracy of 0.876. Finally, Logistic Regression had the worse accuracy with a score of 0.75. However, Naive Bayes had the best precision by far with a score of 0.231 and Logistic regression had the highest recall of 0.29. However, accuracy might not be the best metric to use. As mentioned previously, the dataset is very imbalanced with 93% of the data classified as class 0 (not struggling to pay the loan). Because of this a model that predicts every applicant as class 0 would still perform well. That might be what happened with K-Nearest Neighbors due to its low precision and recall.

Overall, we believe Naive Bayes would be our best model due to it having high precision and recall. We want our model to be precise but also be sensitive. In this case, we want to prioritize high recall since false negatives are what we want to minimize. A false negative in this case means giving a loan out to a person who will struggle. This results in a loss for the bank which we want to minimize. However, we still do want a high precision since it helps to counteract that loss. High precision means reducing the number of false positives, so we want to label applicants as struggling if they really will struggle. Since Naive Bayes has a good balance of both precision and recall as well as accuracy, we believe it is our best model.