AI vs. Non-AI Careers

Final Report

Kelly, Sabrina, Makenzie

October 15, 2025

from pyspark.sql import SparkSession  
from pyspark.sql.functions import col, lit, concat\_ws, lower, regexp\_replace, when, mean as \_mean  
from pyspark.sql import SparkSession, Row  
from pyspark.sql.types import DoubleType  
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler  
from pyspark.ml.classification import RandomForestClassifier  
from pyspark.ml.evaluation import BinaryClassificationEvaluator  
from pyspark.ml import Pipeline  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
  
# --- Start Spark ---  
spark = SparkSession.builder.appName("AI\_Job\_Comparison").getOrCreate()  
  
# --- Load data ---  
df = (spark.read.option("header", "true")  
 .option("inferSchema", "true")  
 .option("multiLine", "true")  
 .option("escape", "\"")  
 .csv("data/lightcast\_job\_postings.csv"))  
  
# --- Create text features for AI labeling ---  
text\_cols = ["TITLE\_RAW", "TITLE\_CLEAN", "BODY", "SKILLS\_NAME",  
 "COMMON\_SKILLS\_NAME", "SPECIALIZED\_SKILLS\_NAME",   
 "SOFTWARE\_SKILLS\_NAME"]  
  
df = df.fillna("")  
df = df.withColumn("text\_features",  
 concat\_ws(" ", \*[lower(regexp\_replace(col(c), r"[^a-zA-Z0-9 ]", "")) for c in text\_cols]))  
  
# --- AI job label ---  
ai\_keywords = [  
 r"\bAI\b", r"\bML\b", r"\bLLM\b", r"\bNLP\b",  
 "artificial intelligence", "machine learning", "deep learning",  
 "computer vision", "generative", "gen ai", "chatgpt", r"gpt-\d+",  
 "transformer", "bert", "prompt engineer", "reinforcement learning"  
]  
ai\_pattern = "|".join([f"(?i){k}" for k in ai\_keywords])  
df = df.withColumn("requires\_ai", when(col("text\_features").rlike(ai\_pattern), lit(1)).otherwise(lit(0)))  
  
# --- Create average salary ---  
df = df.withColumn(  
 "SALARY\_AVG",  
 when(col("SALARY") > 0, col("SALARY"))  
 .when(col("SALARY\_FROM").isNotNull() & col("SALARY\_TO").isNotNull(),  
 (col("SALARY\_FROM") + col("SALARY\_TO")) / 2)  
 .when(col("SALARY\_FROM").isNotNull(), col("SALARY\_FROM"))  
 .when(col("SALARY\_TO").isNotNull(), col("SALARY\_TO"))  
 .otherwise(None)  
)  
  
# --- Handle education level ---  
df = df.withColumn("MIN\_EDULEVELS", when(col("MIN\_EDULEVELS") == 99, 0).otherwise(col("MIN\_EDULEVELS")))  
  
# --- Select relevant features ---  
model\_df = df.select(  
 "requires\_ai",  
 "SALARY\_AVG",  
 "MIN\_YEARS\_EXPERIENCE",  
 "MIN\_EDULEVELS",  
 "STATE",  
 "REMOTE\_TYPE"  
)  
  
# --- Fill numeric nulls ---  
numeric\_cols = ["SALARY\_AVG", "MIN\_YEARS\_EXPERIENCE", "MIN\_EDULEVELS"]  
for c in numeric\_cols:  
 model\_df = model\_df.withColumn(c, col(c).cast(DoubleType()))  
 mean\_val = model\_df.select(\_mean(col(c))).first()[0]  
 model\_df = model\_df.na.fill({c: mean\_val})

[Stage 54:> (0 + 1) / 1] [Stage 55:> (0 + 1) / 1] [Stage 58:> (0 + 1) / 1] [Stage 61:> (0 + 1) / 1]

# --- Encode categorical variables ---  
categorical\_cols = ["STATE", "REMOTE\_TYPE"]  
indexers = [StringIndexer(inputCol=c, outputCol=f"{c}\_IDX", handleInvalid="keep") for c in categorical\_cols]  
encoders = [OneHotEncoder(inputCol=f"{c}\_IDX", outputCol=f"{c}\_VEC") for c in categorical\_cols]  
  
# --- Assemble features ---  
assembler = VectorAssembler(  
 inputCols=numeric\_cols + [f"{c}\_VEC" for c in categorical\_cols],  
 outputCol="features"  
)  
  
# --- Random Forest classifier ---  
rf = RandomForestClassifier(featuresCol="features", labelCol="requires\_ai", numTrees=50, maxDepth=5, seed=42)  
  
# --- Build pipeline ---  
pipeline = Pipeline(stages=indexers + encoders + [assembler, rf])  
  
# --- Split train/test ---  
train\_df, test\_df = model\_df.randomSplit([0.8, 0.2], seed=42)  
  
# --- Train model ---  
rf\_model = pipeline.fit(train\_df)

[Stage 64:> (0 + 1) / 1] [Stage 70:> (0 + 1) / 1] [Stage 76:> (0 + 1) / 1] [Stage 79:> (0 + 1) / 1] [Stage 80:> (0 + 1) / 1] [Stage 81:> (0 + 1) / 1] [Stage 83:> (0 + 1) / 1]

# --- Predict & evaluate ---  
predictions = rf\_model.transform(test\_df)  
evaluator = BinaryClassificationEvaluator(labelCol="requires\_ai", metricName="areaUnderROC")  
roc\_auc = evaluator.evaluate(predictions)  
print(f"ROC-AUC: {roc\_auc:.3f}")

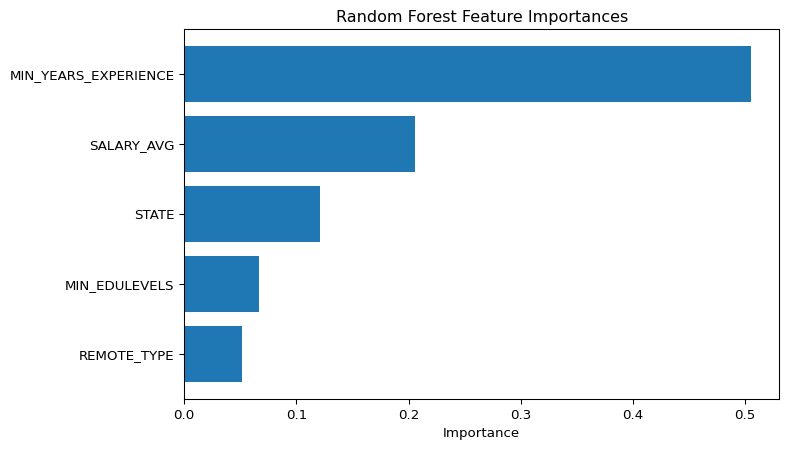
[Stage 93:> (0 + 1) / 1]

ROC-AUC: 0.646

ROC-AUC was used instead of accuracy because it is more suited for binary classifications such as AI vs non-AI.

# --- Feature importances (aggregate categorical vectors) ---  
rf\_stage = rf\_model.stages[-1]  
importances = rf\_stage.featureImportances.toArray()  
  
# Get the sizes of one-hot encoded vectors  
ohe\_stages = [s for s in rf\_model.stages if isinstance(s, OneHotEncoder)]  
categorical\_sizes = []  
for ohe in ohe\_stages:  
 # Number of categories = size of metadata after encoding  
 categorical\_sizes.append(len(ohe.getOutputCol() + "\_metadata") if False else ohe.getOutputCols() if hasattr(ohe,'getOutputCols') else ohe.categorySizes[0] if hasattr(ohe,'categorySizes') else None)  
  
# Alternative simpler approach: we can get vector size from the first row of transformed vector  
features\_vec = rf\_model.transform(test\_df).select("features").first()[0]  
vector\_size = len(features\_vec)  
  
# Approximate: numeric features = 1, categorical features = remaining size divided equally  
num\_numeric = len(numeric\_cols)  
num\_categorical = len(categorical\_cols)  
categorical\_vector\_sizes = [(vector\_size - num\_numeric)//num\_categorical]\*num\_categorical  
  
feature\_sizes = [1]\*num\_numeric + categorical\_vector\_sizes  
feature\_names = numeric\_cols + categorical\_cols  
  
# Aggregate importances  
agg\_importances = []  
start = 0  
for size in feature\_sizes:  
 agg\_importances.append(np.sum(importances[start:start+size]))  
 start += size  
  
# --- Create DataFrame & plot ---  
importance\_df = pd.DataFrame({  
 "Feature": feature\_names,  
 "Importance": agg\_importances  
}).sort\_values("Importance", ascending=False)  
  
plt.figure(figsize=(8,5))  
plt.barh(importance\_df["Feature"], importance\_df["Importance"])  
plt.xlabel("Importance")  
plt.title("Random Forest Feature Importances")  
plt.gca().invert\_yaxis()  
plt.show()

[Stage 102:> (0 + 1) / 1]



# --- Predict sample job listings using the trained pipeline ---  
from pyspark.ml.functions import vector\_to\_array  
  
sample\_jobs = [  
 Row(MIN\_YEARS\_EXPERIENCE=15.0, SALARY\_AVG=300000.0, STATE="NY", MIN\_EDULEVELS=4.0, REMOTE\_TYPE="Remote"),  
 Row(MIN\_YEARS\_EXPERIENCE=2.0, SALARY\_AVG=75000.0, STATE="DE", MIN\_EDULEVELS=0.0, REMOTE\_TYPE="Onsite"),  
 Row(MIN\_YEARS\_EXPERIENCE=3.0, SALARY\_AVG=95000.0, STATE="ND", MIN\_EDULEVELS=1.0, REMOTE\_TYPE="Hybrid")  
]  
  
sample\_df = spark.createDataFrame(sample\_jobs)  
  
# Ensure same schema types  
  
for c in ["MIN\_YEARS\_EXPERIENCE", "SALARY\_AVG", "MIN\_EDULEVELS"]:  
 sample\_df = sample\_df.withColumn(c, col(c).cast(DoubleType()))  
  
# --- Use the \*same pipeline\* to transform data ---  
  
predictions = rf\_model.transform(sample\_df)  
predictions\_array = predictions.withColumn("prob\_array", vector\_to\_array(col("probability")))  
  
# --- Extract probability of AI class and display results ---  
  
results = predictions\_array.select(  
 "MIN\_YEARS\_EXPERIENCE",  
 "SALARY\_AVG",  
 "STATE",  
 "MIN\_EDULEVELS",  
 "REMOTE\_TYPE",  
 col("prob\_array")[1].alias("AI\_prob"),  
 "prediction"  
)  
  
results.show(truncate=False)

+--------------------+----------+-----+-------------+-----------+-------------------+----------+  
|MIN\_YEARS\_EXPERIENCE|SALARY\_AVG|STATE|MIN\_EDULEVELS|REMOTE\_TYPE|AI\_prob |prediction|  
+--------------------+----------+-----+-------------+-----------+-------------------+----------+  
|15.0 |300000.0 |NY |4.0 |Remote |0.3676113258332611 |0.0 |  
|2.0 |75000.0 |DE |0.0 |Onsite |0.19717699753401854|0.0 |  
|3.0 |95000.0 |ND |1.0 |Hybrid |0.19611136141019866|0.0 |  
+--------------------+----------+-----+-------------+-----------+-------------------+----------+

# Random Forest Regression Summary

This analysis used a Random Forest model to distinguish AI from non-AI job postings based on five features: minimum years of experience, average salary, state, minimum education level, and remote work type. These features were chosen because AI roles typically require more experience, offer higher compensation, demand advanced degrees, and are concentrated in specific regions, while remote flexibility can influence applicant reach. The model revealed that years of experience and salary were the most influential factors, followed by state, education, and remote type, reflecting the expected patterns of AI job characteristics.

However, when using this model to predict whether or not a posting would be AI or not, we encountered several limitations that may explain its underperformance. Treating state as a numeric variable may not fully capture the categorical differences between regions, and the model’s majority-vote mechanism can classify borderline AI roles as non-AI, especially when AI postings are underrepresented in the training data. Additional factors such as inconsistent labeling of AI roles, limited sample size, and high variance in salary or education requirements could further reduce predictive accuracy. Despite these challenges, the analysis highlights which features are most informative for distinguishing AI roles and provides a foundation for future models.