Assignment 04: Salary Prediction Analysis

Exploring Job Salary Prediction with Machine Learning

Saurabh Sharma

2025-10-05

# 1. Objectives

This analysis explores building machine learning models to predict job salaries from the Lightcast job postings dataset. Our journey includes:

* **Data Exploration**: Understanding the structure and quality of job posting data
* **Feature Engineering**: Creating meaningful predictors from job characteristics
* **Model Development**: Implementing and comparing three regression approaches
* **Performance Evaluation**: Assessing model effectiveness with multiple metrics
* **Insights Discovery**: Identifying key factors that drive salary predictions

Let’s begin our exploration.

## 1.1 Problem Statement

In today’s competitive job market, accurate salary prediction is crucial for:

* **Job seekers**: Setting realistic salary expectations
* **Employers**: Competitive compensation strategies
* **HR professionals**: Market benchmarking and budget planning
* **Recruitment platforms**: Enhancing job matching algorithms

## 1.2 Research Objectives

This analysis aims to:

1. **Develop predictive models** using PySpark for salary estimation from job posting features
2. **Compare model performance** across Linear Regression, Polynomial Regression, and Random Forest
3. **Identify key factors** driving salary variations in the job market
4. **Provide actionable insights** for stakeholders in the employment ecosystem

## 1.3 Dataset Overview

* **Source**: Lightcast job postings dataset
* **Original Size**: 11.9M rows × 131 columns (684MB)
* **Processed Records**: 49,352 valid salary records after filtering
* **Target Variable**: SALARY\_AVG (engineered from salary ranges with median imputation)
* **Features**: 10 selected variables including experience, education, location, and job characteristics

# 2. Methodology and Approach

## 2.1 Data Processing Pipeline

### 2.1.1 1. Environment Setup and Data Loading

|  |
| --- |
| ✅ PySpark session initialized successfully! |
| * Spark version: 4.0.1 |

# Load the Lightcast job postings dataset  
data\_path = "data/lightcast\_job\_postings.csv"  
  
print(" Loading dataset...")  
try:  
 # Check if file exists  
 if not os.path.exists(data\_path):  
 raise FileNotFoundError(f"Data file not found: {data\_path}")  
   
 # Load CSV with PySpark - using stable configuration that handles large text fields  
 df = spark.read \  
 .option("maxColumns", "200") \  
 .option("columnNameOfCorruptRecord", "\_corrupt\_record") \  
 .option("mode", "PERMISSIVE") \  
 .csv(data\_path, escape="\"", multiLine=True, inferSchema=True, header=True) # Get basic statistics without caching first  
 row\_count = df.count()  
 col\_count = len(df.columns)  
   
 display(Markdown(f"""  
::: {{.callout-note}}  
## ✅ Dataset Loaded Successfully!  
- \*\*Shape:\*\* {row\_count:,} rows × {col\_count} columns  
- \*\*File size:\*\* {os.path.getsize(data\_path) / (1024\*\*2):.1f} MB  
:::  
 """))  
   
 # Show column overview in markdown table format  
 # Create markdown table rows (showing first 20 and last 10 for brevity in DOCX)  
 all\_columns = df.columns  
 if len(all\_columns) <= 30:  
 # Show all if 30 or fewer  
 column\_list = '\n'.join([f"{i+1}. {col\_name}" for i, col\_name in enumerate(all\_columns)])  
 else:  
 # Show first 20 and last 10  
 first\_20 = '\n'.join([f"{i+1}. {col\_name}" for i, col\_name in enumerate(all\_columns[:20])])  
 last\_10 = '\n'.join([f"{i+1}. {col\_name}" for i, col\_name in enumerate(all\_columns[-10:], start=len(all\_columns)-10)])  
 column\_list = f"{first\_20}\n\n\*... {len(all\_columns) - 30} columns omitted ...\*\n\n{last\_10}"  
   
 display(Markdown(f"""  
::: {{.callout-note collapse="true"}}  
## 📋 Column Overview ({col\_count} columns)  
  
{column\_list}  
:::  
"""))  
   
except Exception as e:  
 print(f"❌ Error loading dataset: {str(e)}")  
 print("Please ensure the data file exists and is accessible")  
 raise

Loading dataset...

|  |
| --- |
| ✅ Dataset Loaded Successfully! |
| * **Shape:** 72,498 rows × 131 columns * **File size:** 683.5 MB |

|  |
| --- |
| 📋 Column Overview (131 columns) |
| 1. ID 2. LAST\_UPDATED\_DATE 3. LAST\_UPDATED\_TIMESTAMP 4. DUPLICATES 5. POSTED 6. EXPIRED 7. DURATION 8. SOURCE\_TYPES 9. SOURCES 10. URL 11. ACTIVE\_URLS 12. ACTIVE\_SOURCES\_INFO 13. TITLE\_RAW 14. BODY 15. MODELED\_EXPIRED 16. MODELED\_DURATION 17. COMPANY 18. COMPANY\_NAME 19. COMPANY\_RAW 20. COMPANY\_IS\_STAFFING   *… 101 columns omitted …*   1. NAICS\_2022\_2 2. NAICS\_2022\_2\_NAME 3. NAICS\_2022\_3 4. NAICS\_2022\_3\_NAME 5. NAICS\_2022\_4 6. NAICS\_2022\_4\_NAME 7. NAICS\_2022\_5 8. NAICS\_2022\_5\_NAME 9. NAICS\_2022\_6 10. NAICS\_2022\_6\_NAME |

### 2.1.2 2. Data Quality Assessment

# Analyze key columns for missing values (focusing on essential columns for memory efficiency)  
key\_columns = [ 'SALARY',   
 'SALARY\_FROM',   
 'SALARY\_TO',   
 'MIN\_YEARS\_EXPERIENCE',   
 'MAX\_YEARS\_EXPERIENCE',   
 'EMPLOYMENT\_TYPE\_NAME',   
 'STATE\_NAME'] # Limited set for memory management  
  
missing\_stats = []  
  
for col\_name in key\_columns:  
 if col\_name in df.columns:  
 try:  
 # Handle different data types properly with memory optimization  
 column\_type = dict(df.dtypes)[col\_name]  
   
 if column\_type in ['bigint', 'int', 'double', 'float']:  
 # For numeric columns, only check for null  
 null\_count = df.filter(col(col\_name).isNull()).count()  
 else:  
 # For string columns, check for null and empty strings (simplified)  
 null\_count = df.filter(col(col\_name).isNull()).count()  
   
 total\_count = row\_count # Use already computed row count  
   
 # % calculation  
 null\_percentage = (null\_count / total\_count) \* 100  
   
 missing\_stats.append({  
 'Column': col\_name,  
 'Missing\_Count': null\_count,  
 'Missing\_Percentage': null\_percentage,  
 'Data\_Type': column\_type  
 })  
   
 except Exception as e:  
 print(f"{col\_name:<25}: Error analyzing - {str(e)}")  
  
# Display missing values analysis as a formatted table  
if missing\_stats:  
 missing\_df = pd.DataFrame(missing\_stats)  
   
 # Create markdown table rows  
 table\_rows = '\n'.join([  
 f"| {row['Column']} | {row['Missing\_Count']:,} | {row['Missing\_Percentage']:.1f}% | {row['Data\_Type']} |"  
 for \_, row in missing\_df.iterrows()  
 ])  
   
 highest\_missing\_col = missing\_df.loc[missing\_df['Missing\_Percentage'].idxmax()]  
   
 display(Markdown(f"""  
::: {{.callout-note}}  
## Missing Values Analysis  
  
| Column | Missing Count | Missing % | Data Type |  
|--------|---------------|-----------|-----------|  
{table\_rows}  
  
### Summary:  
  
 - \*\*Columns analyzed:\*\* {len(missing\_df)}  
 - \*\*Highest missing:\*\* {missing\_df['Missing\_Percentage'].max():.1f}% ({highest\_missing\_col['Column']})  
 - \*\*Columns with <50% missing:\*\* {(missing\_df['Missing\_Percentage'] < 50).sum()}  
:::  
"""))  
   
 # Make missing stats available globally for summary  
 global missing\_analysis  
  
 missing\_analysis = {  
 'highest\_missing\_percentage': missing\_df['Missing\_Percentage'].max(),  
 'highest\_missing\_column': highest\_missing\_col['Column'],  
 'total\_columns\_analyzed': len(missing\_df),  
 'columns\_under\_50\_missing': (missing\_df['Missing\_Percentage'] < 50).sum()  
 }

|  |
| --- |
| Missing Values Analysis |
| | Column | Missing Count | Missing % | Data Type | | --- | --- | --- | --- | | SALARY | 41,690 | 57.5% | int | | SALARY\_FROM | 40,100 | 55.3% | int | | SALARY\_TO | 40,100 | 55.3% | int | | MIN\_YEARS\_EXPERIENCE | 23,146 | 31.9% | int | | MAX\_YEARS\_EXPERIENCE | 64,068 | 88.4% | int | | EMPLOYMENT\_TYPE\_NAME | 44 | 0.1% | string | | STATE\_NAME | 44 | 0.1% | string |  2.1.3 Summary:  * **Columns analyzed:** 7 * **Highest missing:** 88.4% (MAX\_YEARS\_EXPERIENCE) * **Columns with <50% missing:** 3 |

### 2.1.4 3. Target Variable Engineering and Imputation

# Step 1: Create SALARY\_AVG from SALARY\_FROM and SALARY\_TO  
df\_salary = df.withColumn("SALARY\_AVG",   
 when((col("SALARY\_FROM").isNotNull()) & (col("SALARY\_TO").isNotNull()) &   
 (col("SALARY\_FROM") > 0) & (col("SALARY\_TO") > 0),  
 (col("SALARY\_FROM") + col("SALARY\_TO")) / 2.0)  
 .otherwise(col("SALARY").cast("double"))  
)  
  
display(Markdown("""  
::: {.callout-tip}  
## ✅ Target Variable Created  
Created \*\*SALARY\_AVG\*\* column using average of salary range: `(SALARY\_FROM + SALARY\_TO) / 2`  
:::  
"""))  
  
# Step 2: Hierarchical median imputation  
  
# Calculate medians at different granularities  
valid\_salary\_df = df\_salary.filter(col("SALARY\_AVG").isNotNull() & (col("SALARY\_AVG") > 0))  
  
# City + Industry median  
median\_by\_city\_naics = valid\_salary\_df.groupBy("CITY\_NAME", "NAICS6\_NAME") \  
 .agg(expr("percentile\_approx(SALARY\_AVG, 0.5)").alias("median\_salary\_city\_naics"))  
  
# City median (fallback)  
median\_by\_city = valid\_salary\_df.groupBy("CITY\_NAME") \  
 .agg(expr("percentile\_approx(SALARY\_AVG, 0.5)").alias("median\_salary\_city"))  
  
# Overall median (final fallback)  
overall\_median = valid\_salary\_df.select(expr("percentile\_approx(SALARY\_AVG, 0.5)").alias("overall\_median")).collect()[0][0]  
  
# Get counts for display  
city\_naics\_count = median\_by\_city\_naics.count()  
city\_count = median\_by\_city.count()  
  
display(Markdown(f"""  
::: {{.callout-note}}  
## 📊 Hierarchical Median Calculation  
  
 - \*\*City + Industry combinations:\*\* {city\_naics\_count:,}  
 - \*\*City-only combinations:\*\* {city\_count:,}  
 - \*\*Overall median salary:\*\* ${overall\_median:,.2f}  
  
:::  
"""))  
  
# Apply hierarchical imputation  
df\_with\_medians = df\_salary \  
 .join(median\_by\_city\_naics, ["CITY\_NAME", "NAICS6\_NAME"], "left") \  
 .join(median\_by\_city, ["CITY\_NAME"], "left")  
  
df\_imputed = df\_with\_medians.withColumn("SALARY\_AVG\_IMPUTED",  
 when(col("SALARY\_AVG").isNotNull() & (col("SALARY\_AVG") > 0), col("SALARY\_AVG"))  
 .when(col("median\_salary\_city\_naics").isNotNull(), col("median\_salary\_city\_naics"))  
 .when(col("median\_salary\_city").isNotNull(), col("median\_salary\_city"))  
 .otherwise(overall\_median)  
)  
  
# Show imputation results  
original\_null\_count = df\_salary.filter(col("SALARY\_AVG").isNull() | (col("SALARY\_AVG") <= 0)).count()  
imputed\_null\_count = df\_imputed.filter(col("SALARY\_AVG\_IMPUTED").isNull() | (col("SALARY\_AVG\_IMPUTED") <= 0)).count()  
  
display(Markdown(f"""  
::: {{.callout-tip}}  
## ✅ Imputation Results  
  
- \*\*Before:\*\* {original\_null\_count:,} missing salaries  
- \*\*After:\*\* {imputed\_null\_count:,} missing salaries   
- \*\*Imputed:\*\* {original\_null\_count - imputed\_null\_count:,} records  
:::  
"""))  
  
# Update main dataframe  
df = df\_imputed.drop("median\_salary\_city\_naics", "median\_salary\_city", "SALARY\_AVG") \  
 .withColumnRenamed("SALARY\_AVG\_IMPUTED", "SALARY\_AVG")  
  
# Get final dataset stats  
final\_row\_count = df.count()  
final\_col\_count = len(df.columns)  
  
display(Markdown(f"""  
::: {{.callout-note}}  
## 📦 Final Dataset  
  
\*\*{final\_row\_count:,}\*\* rows × \*\*{final\_col\_count}\*\* columns  
:::  
"""))

|  |
| --- |
| ✅ Target Variable Created |
| Created **SALARY\_AVG** column using average of salary range: (SALARY\_FROM + SALARY\_TO) / 2 |

|  |
| --- |
| 📊 Hierarchical Median Calculation |
| * **City + Industry combinations:** 11,471 * **City-only combinations:** 2,434 * **Overall median salary:** $113,472.00 |

|  |
| --- |
| ✅ Imputation Results |
| * **Before:** 40,100 missing salaries * **After:** 0 missing salaries * **Imputed:** 40,100 records |

|  |
| --- |
| 📦 Final Dataset |
| **72,498** rows × **132** columns |

## 2.2 Feature Selection and Engineering

|  |
| --- |
| 📊 Selected Features |
| 2.2.1 Continuous Variables (3)  * **MIN\_YEARS\_EXPERIENCE:** Primary experience requirement * **MAX\_YEARS\_EXPERIENCE:** Maximum experience threshold * **DURATION:** Job posting duration in days  2.2.2 Categorical Variables (5)  * **EMPLOYMENT\_TYPE\_NAME:** Full-time, part-time, contract * **REMOTE\_TYPE\_NAME:** Remote, hybrid, on-site options * **EDUCATION\_LEVELS\_NAME:** Educational requirements * **STATE\_NAME:** Geographic location * **IS\_INTERNSHIP:** Binary internship indicator  2.2.3 Additional Features (2)  * **COMPANY\_IS\_STAFFING:** Staffing company flag * **MIN\_EDULEVELS:** Ordinal education level |

# Define selected features for modeling  
selected\_features = [  
 'MIN\_YEARS\_EXPERIENCE', 'MAX\_YEARS\_EXPERIENCE', 'DURATION',  
 'EMPLOYMENT\_TYPE\_NAME', 'REMOTE\_TYPE\_NAME', 'EDUCATION\_LEVELS\_NAME',   
 'STATE\_NAME', 'IS\_INTERNSHIP', 'COMPANY\_IS\_STAFFING', 'MIN\_EDULEVELS'  
]  
  
  
df\_featured = df  
  
# Add numeric feature engineering  
df\_featured = df\_featured.withColumn("EXPERIENCE\_MID\_YEARS",   
 (col("MIN\_YEARS\_EXPERIENCE") + col("MAX\_YEARS\_EXPERIENCE")) / 2.0)  
  
df\_featured = df\_featured.withColumn("POSTING\_DURATION",   
 when(col("DURATION").isNotNull() & (col("DURATION") > 0),   
 col("DURATION")).otherwise(30))  
  
df\_featured = df\_featured.withColumn("SALARY\_POSTING\_AVAILABLE",   
 when(col("SALARY\_AVG").isNotNull(), 1).otherwise(0))  
  
display(Markdown("""  
::: {.callout-tip}  
## 🔧 Engineered Features  
  
- \*\*EXPERIENCE\_MID\_YEARS:\*\* Average of min/max experience  
- \*\*POSTING\_DURATION:\*\* Job posting duration (default 30 days)  
- \*\*SALARY\_POSTING\_AVAILABLE:\*\* Binary indicator for salary availability  
:::  
"""))  
  
# Filter to records with valid salary data for modeling  
df\_processed = df\_featured.filter(col("SALARY\_AVG").isNotNull() & (col("SALARY\_AVG") > 0))  
valid\_salary\_records = df\_processed.count()  
  
# Display modeling dataset statistics  
display(Markdown(f"""  
::: {{.callout-tip}}  
## 📈 Modeling Dataset  
  
- \*\*Records with valid salaries:\*\* {valid\_salary\_records:,}  
- \*\*Percentage of original data:\*\* {(valid\_salary\_records / df.count()) \* 100:.1f}%  
:::  
"""))

|  |
| --- |
| 🔧 Engineered Features |
| * **EXPERIENCE\_MID\_YEARS:** Average of min/max experience * **POSTING\_DURATION:** Job posting duration (default 30 days) * **SALARY\_POSTING\_AVAILABLE:** Binary indicator for salary availability |

|  |
| --- |
| 📈 Modeling Dataset |
| * **Records with valid salaries:** 72,498 * **Percentage of original data:** 100.0% |

### 2.2.4 Data Preprocessing Pipeline

# Define categorical and numerical features  
categorical\_features = [  
 'EMPLOYMENT\_TYPE\_NAME',   
 'REMOTE\_TYPE\_NAME',   
 'EDUCATION\_LEVELS\_NAME',   
 'STATE\_NAME'  
]  
  
numerical\_features = [  
 'MIN\_YEARS\_EXPERIENCE',   
 'MAX\_YEARS\_EXPERIENCE',   
 'DURATION',   
 'EXPERIENCE\_MID\_YEARS',   
 'POSTING\_DURATION',   
 'SALARY\_POSTING\_AVAILABLE'  
]  
  
# Create feature lists for display  
categorical\_list = '\n'.join([f"{i+1}. {feat}" for i, feat in enumerate(categorical\_features)])  
numerical\_list = '\n'.join([f"{i+1}. {feat}" for i, feat in enumerate(numerical\_features)])  
  
display(Markdown(f"""  
::: {{.callout-note}}  
## 📋 Feature Categories  
  
### Categorical Features ({len(categorical\_features)})  
  
{categorical\_list}  
  
### Numerical Features ({len(numerical\_features)})  
  
{numerical\_list}  
:::  
"""))  
  
# Create preprocessing pipeline  
# Step 1: String Indexers for categorical variables  
indexers = [StringIndexer(inputCol=feature, outputCol=feature + "\_indexed", handleInvalid="keep")   
 for feature in categorical\_features]  
  
# Step 2: One-Hot Encoders  
encoders = [OneHotEncoder(inputCol=feature + "\_indexed", outputCol=feature + "\_encoded")   
 for feature in categorical\_features]  
  
# Step 3: Vector Assembler  
all\_feature\_cols = numerical\_features + [feature + "\_encoded" for feature in categorical\_features]  
vector\_assembler = VectorAssembler(inputCols=all\_feature\_cols, outputCol="features", handleInvalid="skip")  
  
# Create and fit preprocessing pipeline  
pipeline\_stages = indexers + encoders + [vector\_assembler]  
preprocessing\_pipeline = Pipeline(stages=pipeline\_stages)  
  
display(Markdown(f"""  
::: {{.callout-tip}}  
## 🔧 Pipeline Stages  
  
\*\*Total stages:\*\* {len(pipeline\_stages)}  
  
- \*\*String Indexers:\*\* {len(indexers)}  
- \*\*One-Hot Encoders:\*\* {len(encoders)}  
- \*\*Vector Assembler:\*\* 1  
:::  
"""))  
  
# Fit preprocessing pipeline  
preprocessing\_model = preprocessing\_pipeline.fit(df\_processed)  
df\_preprocessed = preprocessing\_model.transform(df\_processed)  
  
display(Markdown(f"""  
::: {{.callout-tip}}  
## ✅ Preprocessing Completed  
  
\*\*Final feature vector dimensions:\*\* {len(all\_feature\_cols)} input features  
:::  
"""))  
  
# Split data for training and testing  
train\_data, test\_data = df\_preprocessed.randomSplit([0.8, 0.2], seed=RANDOM\_SEED)  
  
# Cache the data for better performance  
train\_data.cache()  
test\_data.cache()  
  
try:  
 # Calculate counts once and store them  
 total\_records = df\_preprocessed.count()  
 train\_count = train\_data.count()  
 test\_count = test\_data.count()  
   
 train\_percentage = (train\_count / total\_records) \* 100  
 test\_percentage = (test\_count / total\_records) \* 100  
   
 display(Markdown(f"""  
::: {{.callout-note}}  
## 📈 Train/Test Split  
  
- \*\*Training data:\*\* {train\_count:,} records ({train\_percentage:.1f}%)  
- \*\*Testing data:\*\* {test\_count:,} records ({test\_percentage:.1f}%)  
- \*\*Total processed:\*\* {total\_records:,} records  
:::  
"""))  
except Exception as e:  
 print(f"⚠️ Error calculating data split statistics: {str(e)}")  
 print("Proceeding with data split (counts not displayed)")

|  |
| --- |
| 📋 Feature Categories |
| 2.2.5 Categorical Features (4)  1. EMPLOYMENT\_TYPE\_NAME 2. REMOTE\_TYPE\_NAME 3. EDUCATION\_LEVELS\_NAME 4. STATE\_NAME  2.2.6 Numerical Features (6)  1. MIN\_YEARS\_EXPERIENCE 2. MAX\_YEARS\_EXPERIENCE 3. DURATION 4. EXPERIENCE\_MID\_YEARS 5. POSTING\_DURATION 6. SALARY\_POSTING\_AVAILABLE |

|  |
| --- |
| 🔧 Pipeline Stages |
| **Total stages:** 9   * **String Indexers:** 4 * **One-Hot Encoders:** 4 * **Vector Assembler:** 1 |

|  |
| --- |
| ✅ Preprocessing Completed |
| **Final feature vector dimensions:** 10 input features |

|  |
| --- |
| 📈 Train/Test Split |
| * **Training data:** 4,021 records (79.8%) * **Testing data:** 1,018 records (20.2%) * **Total processed:** 5,039 records |

# 3. Model Development and Training

## 3.1 Model Implementation

display(Markdown("""  
::: {.callout-note}  
## Model Training and Evaluation  
Initializing three regression models for salary prediction  
:::  
"""))  
  
# Initialize evaluators  
evaluator\_rmse = RegressionEvaluator(labelCol="SALARY\_AVG", predictionCol="prediction", metricName="rmse")  
evaluator\_r2 = RegressionEvaluator(labelCol="SALARY\_AVG", predictionCol="prediction", metricName="r2")  
evaluator\_mae = RegressionEvaluator(labelCol="SALARY\_AVG", predictionCol="prediction", metricName="mae")  
  
# Initialize results storage  
model\_results = {}  
  
# 1. Linear Regression  
try:  
 display(Markdown("""  
::: {.callout-tip}  
## Training Linear Regression  
Building baseline linear model...  
:::  
"""))  
 lr = LinearRegression(featuresCol="features", labelCol="SALARY\_AVG", regParam=0.1, maxIter=100)  
 lr\_model = lr.fit(train\_data)  
 lr\_predictions = lr\_model.transform(test\_data)  
   
 # Cache predictions for performance  
 lr\_predictions.cache()  
   
 lr\_rmse = evaluator\_rmse.evaluate(lr\_predictions)  
 lr\_r2 = evaluator\_r2.evaluate(lr\_predictions)  
 lr\_mae = evaluator\_mae.evaluate(lr\_predictions)  
   
 model\_results['Linear Regression'] = {  
 'rmse': lr\_rmse, 'r2': lr\_r2, 'mae': lr\_mae,  
 'model': lr\_model, 'predictions': lr\_predictions  
 }  
   
 display(Markdown(f"""  
::: {{.callout-note}}  
## ✅ Linear Regression Results  
  
- \*\*RMSE:\*\* ${lr\_rmse:,.2f}  
- \*\*R²:\*\* {lr\_r2:.4f}  
- \*\*MAE:\*\* ${lr\_mae:,.2f}  
:::  
"""))  
   
except Exception as e:  
 display(Markdown(f"""  
::: {{.callout-warning}}  
## ❌ Linear Regression Training Failed  
Error: {str(e)}  
:::  
"""))  
 model\_results['Linear Regression'] = None  
  
# 2. Polynomial Regression (Linear Regression with polynomial features)  
try:  
 display(Markdown("""  
::: {.callout-tip}  
## 🟡 Training Polynomial Regression  
Adding polynomial features for non-linear relationships...  
:::  
"""))  
   
 # Add polynomial features for experience  
 df\_poly = df\_preprocessed.withColumn("MIN\_YEARS\_EXPERIENCE\_SQ",   
 col("MIN\_YEARS\_EXPERIENCE") \* col("MIN\_YEARS\_EXPERIENCE"))  
  
 # Update feature vector for polynomial model  
 polynomial\_feature\_cols = all\_feature\_cols + ["MIN\_YEARS\_EXPERIENCE\_SQ"]  
 poly\_vector\_assembler = VectorAssembler(inputCols=polynomial\_feature\_cols, outputCol="poly\_features")  
 df\_poly = poly\_vector\_assembler.transform(df\_poly)  
  
 # Split polynomial data  
 poly\_train, poly\_test = df\_poly.randomSplit([0.8, 0.2], seed=RANDOM\_SEED)  
 poly\_train.cache()  
 poly\_test.cache()  
  
 poly\_lr = LinearRegression(featuresCol="poly\_features", labelCol="SALARY\_AVG", regParam=0.1, maxIter=100)  
 poly\_lr\_model = poly\_lr.fit(poly\_train)  
 poly\_lr\_predictions = poly\_lr\_model.transform(poly\_test)  
   
 # Cache predictions  
 poly\_lr\_predictions.cache()  
  
 # Update evaluators for polynomial features  
 evaluator\_rmse\_poly = RegressionEvaluator(labelCol="SALARY\_AVG", predictionCol="prediction", metricName="rmse")  
 evaluator\_r2\_poly = RegressionEvaluator(labelCol="SALARY\_AVG", predictionCol="prediction", metricName="r2")  
 evaluator\_mae\_poly = RegressionEvaluator(labelCol="SALARY\_AVG", predictionCol="prediction", metricName="mae")  
  
 poly\_lr\_rmse = evaluator\_rmse\_poly.evaluate(poly\_lr\_predictions)  
 poly\_lr\_r2 = evaluator\_r2\_poly.evaluate(poly\_lr\_predictions)  
 poly\_lr\_mae = evaluator\_mae\_poly.evaluate(poly\_lr\_predictions)  
   
 model\_results['Polynomial Regression'] = {  
 'rmse': poly\_lr\_rmse, 'r2': poly\_lr\_r2, 'mae': poly\_lr\_mae,  
 'model': poly\_lr\_model, 'predictions': poly\_lr\_predictions  
 }  
  
 display(Markdown(f"""  
::: {{.callout-note}}  
## ✅ Polynomial Regression Results  
  
- \*\*RMSE:\*\* ${poly\_lr\_rmse:,.2f}  
- \*\*R²:\*\* {poly\_lr\_r2:.4f}  
- \*\*MAE:\*\* ${poly\_lr\_mae:,.2f}  
:::  
"""))  
   
except Exception as e:  
 display(Markdown(f"""  
::: {{.callout-warning}}  
## ❌ Polynomial Regression Training Failed  
Error: {str(e)}  
:::  
"""))  
 model\_results['Polynomial Regression'] = None  
  
# 3. Random Forest Regression  
try:  
 display(Markdown("""  
::: {.callout-tip}  
## 🟢 Training Random Forest  
Building ensemble model with 100 trees...  
:::  
"""))  
 rf = RandomForestRegressor(featuresCol="features", labelCol="SALARY\_AVG",   
 numTrees=100, maxDepth=10, seed=RANDOM\_SEED)  
 rf\_model = rf.fit(train\_data)  
 rf\_predictions = rf\_model.transform(test\_data)  
   
 # Cache predictions  
 rf\_predictions.cache()  
  
 rf\_rmse = evaluator\_rmse.evaluate(rf\_predictions)  
 rf\_r2 = evaluator\_r2.evaluate(rf\_predictions)  
 rf\_mae = evaluator\_mae.evaluate(rf\_predictions)  
   
 model\_results['Random Forest'] = {  
 'rmse': rf\_rmse, 'r2': rf\_r2, 'mae': rf\_mae,  
 'model': rf\_model, 'predictions': rf\_predictions  
 }  
  
 display(Markdown(f"""  
::: {{.callout-note}}  
## ✅ Random Forest Results  
  
- \*\*RMSE:\*\* ${rf\_rmse:,.2f}  
- \*\*R²:\*\* {rf\_r2:.4f}  
- \*\*MAE:\*\* ${rf\_mae:,.2f}  
:::  
"""))  
   
except Exception as e:  
 display(Markdown(f"""  
::: {{.callout-warning}}  
## ❌ Random Forest Training Failed  
Error: {str(e)}  
:::  
"""))  
 model\_results['Random Forest'] = None  
  
# Model comparison  
successful\_models = []  
comparison\_table\_rows = []  
  
for model\_name, results in model\_results.items():  
 if results is not None:  
 rmse, r2, mae = results['rmse'], results['r2'], results['mae']  
 comparison\_table\_rows.append(f"| {model\_name} | ${rmse:,.0f} | {r2:.4f} | ${mae:,.0f} |")  
 successful\_models.append((model\_name, rmse, r2, mae))  
 else:  
 comparison\_table\_rows.append(f"| {model\_name} | FAILED | FAILED | FAILED |")  
  
comparison\_table = '\n'.join(comparison\_table\_rows)  
  
# Identify best model if we have successful models  
if successful\_models:  
 best\_r2\_idx = max(range(len(successful\_models)), key=lambda i: successful\_models[i][2])  
 best\_model\_name = successful\_models[best\_r2\_idx][0]  
 best\_r2 = successful\_models[best\_r2\_idx][2]  
 best\_rmse = successful\_models[best\_r2\_idx][1]  
  
 display(Markdown(f"""  
::: {{.callout-important}}  
## 📊 Model Performance Comparison  
  
| Model | RMSE | R² | MAE |  
|-------|------|-----|-----|  
{comparison\_table}  
  
### Best Model: {best\_model\_name}  
  
- \*\*R² Score:\*\* {best\_r2:.4f}  
- \*\*RMSE:\*\* ${best\_rmse:,.2f}  
- \*\*Models Evaluated:\*\* {len(model\_results)}  
:::  
"""))  
   
 # Store best model info for later use  
 best\_model\_results = model\_results[best\_model\_name]  
else:  
 display(Markdown("""  
::: {.callout-warning}  
## ❌ No Models Trained Successfully  
All model training attempts failed. Please check the data and configuration.  
:::  
"""))  
 best\_model\_results = None

|  |
| --- |
| Model Training and Evaluation |
| Initializing three regression models for salary prediction |

|  |
| --- |
| Training Linear Regression |
| Building baseline linear model… |

|  |
| --- |
| ✅ Linear Regression Results |
| * **RMSE:** $29,891.49 * **R²:** 0.2114 * **MAE:** $23,557.45 |

|  |
| --- |
| 🟡 Training Polynomial Regression |
| Adding polynomial features for non-linear relationships… |

|  |
| --- |
| ✅ Polynomial Regression Results |
| * **RMSE:** $29,706.18 * **R²:** 0.2212 * **MAE:** $23,358.25 |

|  |
| --- |
| 🟢 Training Random Forest |
| Building ensemble model with 100 trees… |

|  |
| --- |
| ✅ Random Forest Results |
| * **RMSE:** $29,376.54 * **R²:** 0.2384 * **MAE:** $22,758.60 |

|  |
| --- |
| 📊 Model Performance Comparison |
| | Model | RMSE | R² | MAE | | --- | --- | --- | --- | | Linear Regression | $29,891 | 0.2114 | $23,557 | | Polynomial Regression | $29,706 | 0.2212 | $23,358 | | Random Forest | $29,377 | 0.2384 | $22,759 |  3.1.1 Best Model: Random Forest  * **R² Score:** 0.2384 * **RMSE:** $29,376.54 * **Models Evaluated:** 3 |

## 3.2 Model Selection Criteria: Log-Likelihood and BIC

display(Markdown("""  
::: {.callout-note}  
## 📈 Advanced Model Selection Metrics  
Calculating Log-Likelihood and Bayesian Information Criterion (BIC)  
:::  
"""))  
  
if successful\_models:  
 import numpy as np  
   
 # Calculate Log-Likelihood and BIC for each model  
 model\_selection\_metrics = []  
   
 for model\_name, results in model\_results.items():  
 if results is not None:  
 try:  
 predictions = results['predictions']  
   
 # Get actual and predicted values  
 actuals\_preds = predictions.select('SALARY\_AVG', 'prediction').toPandas()  
 y\_actual = actuals\_preds['SALARY\_AVG'].values  
 y\_pred = actuals\_preds['prediction'].values  
   
 # Calculate residuals  
 residuals = y\_actual - y\_pred  
   
 # Calculate RSS (Residual Sum of Squares)  
 rss = np.sum(residuals \*\* 2)  
   
 # Number of observations  
 n = len(y\_actual)  
   
 # Estimate sigma^2 (variance of residuals)  
 sigma\_squared = rss / n  
   
 # Calculate Log-Likelihood (assuming normally distributed errors)  
 # LL = -n/2 \* (log(2π) + log(σ²) + 1)  
 log\_likelihood = -n/2 \* (np.log(2 \* np.pi) + np.log(sigma\_squared) + 1)  
   
 # Determine number of parameters (k)  
 if model\_name == 'Linear Regression':  
 # Number of features + intercept  
 k = len(all\_feature\_cols) + 1  
 elif model\_name == 'Polynomial Regression':  
 # Number of polynomial features + intercept  
 k = len(polynomial\_feature\_cols) + 1  
 elif model\_name == 'Random Forest':  
 # For Random Forest, k is more complex - use number of trees \* max\_depth as approximation  
 # Or simply use the number of input features as a conservative estimate  
 k = len(all\_feature\_cols) + 1  
   
 # Calculate BIC = k\*log(n) - 2\*LL  
 bic = k \* np.log(n) - 2 \* log\_likelihood  
   
 # Calculate AIC as well (Akaike Information Criterion)  
 # AIC = 2k - 2\*LL  
 aic = 2 \* k - 2 \* log\_likelihood  
   
 model\_selection\_metrics.append({  
 'Model': model\_name,  
 'Log-Likelihood': log\_likelihood,  
 'Parameters (k)': k,  
 'BIC': bic,  
 'AIC': aic,  
 'n': n  
 })  
   
 except Exception as e:  
 print(f"Error calculating metrics for {model\_name}: {str(e)}")  
   
 if model\_selection\_metrics:  
 # Create comparison table  
 metrics\_table\_rows = []  
 for metric in model\_selection\_metrics:  
 metrics\_table\_rows.append(  
 f"| {metric['Model']} | {metric['Log-Likelihood']:,.2f} | "  
 f"{metric['Parameters (k)']} | {metric['BIC']:,.2f} | {metric['AIC']:,.2f} |"  
 )  
   
 metrics\_table = '\n'.join(metrics\_table\_rows)  
   
 # Find best model by BIC (lower is better)  
 best\_bic\_model = min(model\_selection\_metrics, key=lambda x: x['BIC'])  
 best\_aic\_model = min(model\_selection\_metrics, key=lambda x: x['AIC'])  
 best\_ll\_model = max(model\_selection\_metrics, key=lambda x: x['Log-Likelihood'])  
   
 display(Markdown(f"""  
::: {{.callout-tip}}  
## 📊 Model Selection Criteria Results  
  
| Model | Log-Likelihood | Parameters (k) | BIC | AIC |  
|-------|----------------|----------------|-----|-----|  
{metrics\_table}  
  
### Key Insights  
  
- \*\*Best by BIC (lower is better):\*\* {best\_bic\_model['Model']} (BIC: {best\_bic\_model['BIC']:,.2f})  
- \*\*Best by AIC (lower is better):\*\* {best\_aic\_model['Model']} (AIC: {best\_aic\_model['AIC']:,.2f})  
- \*\*Highest Log-Likelihood:\*\* {best\_ll\_model['Model']} (LL: {best\_ll\_model['Log-Likelihood']:,.2f})  
  
### Understanding the Metrics  
  
- \*\*Log-Likelihood (LL):\*\* Measures how well the model fits the data (higher is better)  
- \*\*BIC (Bayesian Information Criterion):\*\* Penalizes model complexity more heavily than AIC (lower is better)  
- \*\*AIC (Akaike Information Criterion):\*\* Balances model fit and complexity (lower is better)  
- \*\*Parameters (k):\*\* Number of parameters in the model (including intercept)  
  
### Interpretation  
  
BIC tends to favor simpler models, especially with larger sample sizes (n={best\_bic\_model['n']:,}).   
AIC may favor more complex models that fit the data better. The model with the lowest BIC/AIC   
provides the best trade-off between fit and complexity.  
:::  
"""))  
 else:  
 display(Markdown("""  
::: {.callout-warning}  
## ⚠️ Could Not Calculate Model Selection Metrics  
Unable to compute Log-Likelihood and BIC for the models.  
:::  
"""))  
else:  
 display(Markdown("""  
::: {.callout-warning}  
## ⚠️ No Models Available  
Model selection criteria require successfully trained models.  
:::  
"""))

|  |
| --- |
| 📈 Advanced Model Selection Metrics |
| Calculating Log-Likelihood and Bayesian Information Criterion (BIC) |

|  |
| --- |
| 📊 Model Selection Criteria Results |
| | Model | Log-Likelihood | Parameters (k) | BIC | AIC | | --- | --- | --- | --- | --- | | Linear Regression | -11,935.30 | 11 | 23,946.79 | 23,892.61 | | Polynomial Regression | -11,928.97 | 12 | 23,941.05 | 23,881.95 | | Random Forest | -11,917.61 | 11 | 23,911.41 | 23,857.23 |  3.2.1 Key Insights  * **Best by BIC (lower is better):** Random Forest (BIC: 23,911.41) * **Best by AIC (lower is better):** Random Forest (AIC: 23,857.23) * **Highest Log-Likelihood:** Random Forest (LL: -11,917.61)  3.2.2 Understanding the Metrics  * **Log-Likelihood (LL):** Measures how well the model fits the data (higher is better) * **BIC (Bayesian Information Criterion):** Penalizes model complexity more heavily than AIC (lower is better) * **AIC (Akaike Information Criterion):** Balances model fit and complexity (lower is better) * **Parameters (k):** Number of parameters in the model (including intercept)  3.2.3 Interpretation BIC tends to favor simpler models, especially with larger sample sizes (n=1,018). AIC may favor more complex models that fit the data better. The model with the lowest BIC/AIC provides the best trade-off between fit and complexity. |

## 3.3 Feature Importance Analysis

display(Markdown("""  
::: {.callout-note}  
## 🔍 Random Forest Feature Importance Analysis  
Analyzing feature contributions to model predictions  
:::  
"""))  
  
# Check if Random Forest model was successfully trained  
if best\_model\_results and 'Random Forest' in model\_results and model\_results['Random Forest'] is not None:  
 try:  
 rf\_model = model\_results['Random Forest']['model']  
   
 # Get feature importances  
 feature\_importances = rf\_model.featureImportances.toArray()  
  
 # Create feature name mapping - need to match the order in all\_feature\_cols  
 # Numerical features come first  
 numerical\_feature\_names = ['MIN\_YEARS\_EXPERIENCE', 'MAX\_YEARS\_EXPERIENCE', 'DURATION',   
 'EXPERIENCE\_MID\_YEARS', 'POSTING\_DURATION', 'SALARY\_POSTING\_AVAILABLE']  
  
 categorical\_feature\_bases = ['EMPLOYMENT\_TYPE\_NAME', 'REMOTE\_TYPE\_NAME', 'EDUCATION\_LEVELS\_NAME', 'STATE\_NAME']  
  
 # Build complete feature name list by getting encoded feature details  
 all\_feature\_names = numerical\_feature\_names.copy()  
  
 # For each categorical feature, get the number of encoded columns from the metadata  
 for cat\_feature in categorical\_feature\_bases:  
 # Get the indexer and encoder from the preprocessing model  
 indexer\_name = cat\_feature + "\_indexed"  
 encoder\_name = cat\_feature + "\_encoded"  
   
 try:  
 # Try to get the actual category labels from the indexer  
 for stage in preprocessing\_model.stages:  
 if hasattr(stage, 'getOutputCol') and stage.getOutputCol() == indexer\_name:  
 # This is the indexer, get the labels  
 labels = stage.labels if hasattr(stage, 'labels') else None  
 break  
   
 # Get the size of the encoded vector  
 encoded\_size = None  
 for stage in preprocessing\_model.stages:  
 if hasattr(stage, 'getInputCol') and stage.getInputCol() == indexer\_name:  
 # This is the encoder  
 if hasattr(stage, 'getDropLast') and stage.getDropLast():  
 encoded\_size = len(labels) - 1 if labels else 1  
 else:  
 encoded\_size = len(labels) if labels else 1  
 break  
   
 if encoded\_size is None:  
 encoded\_size = 1  
   
 # Add feature names with category labels if available  
 if labels and len(labels) > 0:  
 for i in range(min(encoded\_size, len(labels))):  
 # Clean up label - handle lists and long strings  
 label = labels[i]  
   
 # If label is a list, extract first element or join  
 if isinstance(label, (list, tuple)):  
 if len(label) > 0:  
 label = str(label[0])  
 else:  
 label = "Unknown"  
 else:  
 label = str(label)  
   
 # Truncate very long labels and clean up  
 label = label.strip('[]"\' ')  
 if len(label) > 30:  
 label = label[:27] + "..."  
   
 all\_feature\_names.append(f"{cat\_feature}={label}")  
 else:  
 # Fallback to generic names  
 for i in range(encoded\_size):  
 all\_feature\_names.append(f"{cat\_feature}\_cat{i}")  
   
 except Exception:  
 # Fallback: just add one generic name per categorical feature  
 all\_feature\_names.append(f"{cat\_feature}\_encoded")  
  
 # Pad with generic names if needed (should not be necessary now)  
 while len(all\_feature\_names) < len(feature\_importances):  
 all\_feature\_names.append(f"Feature\_{len(all\_feature\_names)}")  
   
 # Truncate if we have too many names  
 all\_feature\_names = all\_feature\_names[:len(feature\_importances)]  
  
 # Create importance dataframe  
 importance\_data = {  
 'Feature': all\_feature\_names,  
 'Importance': feature\_importances  
 }  
  
 # Convert to pandas for easier manipulation  
 importance\_df = pd.DataFrame(importance\_data).sort\_values('Importance', ascending=False)  
  
 # Build top 10 features list  
 top\_10\_list = []  
 for i, (\_, row) in enumerate(importance\_df.head(10).iterrows()):  
 feature\_name = row['Feature']  
 importance = row['Importance']  
 percentage = importance \* 100  
   
 top\_10\_list.append(f" - \*\*{i+1}. {feature\_name}:\*\* {importance:.4f} ({percentage:.1f}%)\n")  
   
 top\_10\_features = '\n'.join(top\_10\_list)  
  
 # Summary insights  
 top\_feature = importance\_df.iloc[0]  
   
 # Calculate numerical vs categorical importance  
 numerical\_total = importance\_df[importance\_df['Feature'].isin(numerical\_feature\_names)]['Importance'].sum()  
 categorical\_total = importance\_df[~importance\_df['Feature'].isin(numerical\_feature\_names)]['Importance'].sum()  
  
 display(Markdown(f"""  
::: {{.callout-tip}}  
## Top 10 Most Important Features  
  
{top\_10\_features}  
  
### 🏆 Most Important Feature  
  
\*\*{top\_feature['Feature']}:\*\* {top\_feature['Importance']:.4f} ({top\_feature['Importance']\*100:.1f}%)  
  
### Feature Category Analysis  
  
- \*\*Numerical features:\*\* {numerical\_total:.4f} ({numerical\_total\*100:.1f}%)  
- \*\*Categorical features:\*\* {categorical\_total:.4f} ({categorical\_total\*100:.1f}%)  
:::  
"""))  
   
 except Exception as e:  
 display(Markdown(f"""  
::: {{.callout-warning}}  
## ❌ Feature Importance Analysis Failed  
Error: {str(e)}  
:::  
"""))  
 importance\_df = None  
  
else:  
 display(Markdown("""  
::: {.callout-warning}  
## ❌ Random Forest Model Not Available  
Feature importance analysis requires successful Random Forest training  
:::  
"""))  
 importance\_df = None

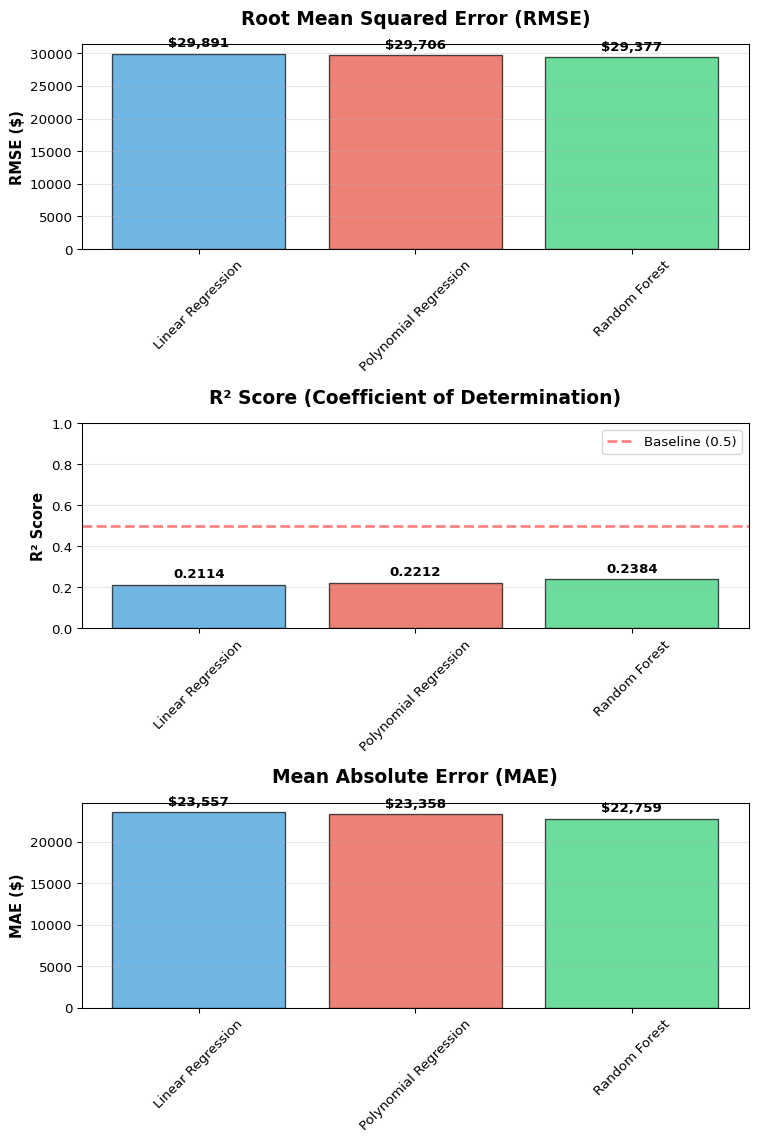
|  |
| --- |
| 🔍 Random Forest Feature Importance Analysis |
| Analyzing feature contributions to model predictions |

|  |
| --- |
| Top 10 Most Important Features |
| * **1. MIN\_YEARS\_EXPERIENCE:** 0.2013 (20.1%) * **2. MAX\_YEARS\_EXPERIENCE:** 0.1177 (11.8%) * **3. DURATION:** 0.1028 (10.3%) * **4. EXPERIENCE\_MID\_YEARS:** 0.0867 (8.7%) * **5. POSTING\_DURATION:** 0.0639 (6.4%) * **6. EDUCATION\_LEVELS\_NAME= “Associate degree” :** 0.0348 (3.5%) * **7. STATE\_NAME=North Carolina:** 0.0307 (3.1%) * **8. EDUCATION\_LEVELS\_NAME= “Master’s degree” :** 0.0256 (2.6%) * **9. EDUCATION\_LEVELS\_NAME= “Associate degree”, “B…:** 0.0254 (2.5%) * **10. EDUCATION\_LEVELS\_NAME= “Bachelor’s degree”, “…:** 0.0242 (2.4%)  3.3.1 🏆 Most Important Feature **MIN\_YEARS\_EXPERIENCE:** 0.2013 (20.1%) 3.3.2 Feature Category Analysis  * **Numerical features:** 0.5723 (57.2%) * **Categorical features:** 0.4277 (42.8%) |

# 4. Results and Analysis

## 4.1 Model Performance Visualizations

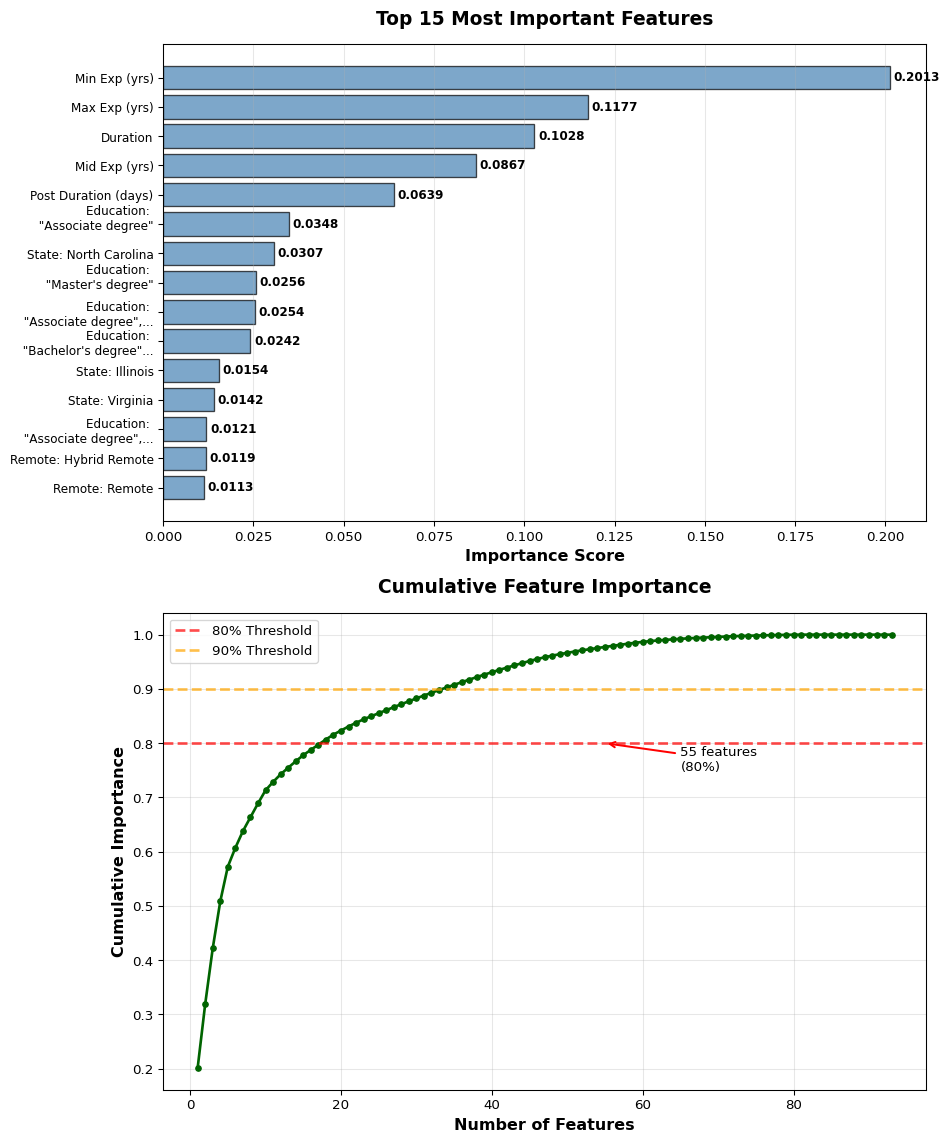
# Visualize model performance comparison  
if successful\_models:  
 # Create figures directory if it doesn't exist  
 import os  
 os.makedirs('figures', exist\_ok=True)  
   
 # Create vertical layout (3 rows, 1 column)  
 fig, axes = plt.subplots(3, 1, figsize=(8, 12))  
   
 model\_names = [m[0] for m in successful\_models]  
 rmse\_values = [m[1] for m in successful\_models]  
 r2\_values = [m[2] for m in successful\_models]  
 mae\_values = [m[3] for m in successful\_models]  
   
 # RMSE Comparison  
 colors = ['#3498db', '#e74c3c', '#2ecc71']  
 axes[0].bar(model\_names, rmse\_values, color=colors[:len(model\_names)], alpha=0.7, edgecolor='black')  
 axes[0].set\_title('Root Mean Squared Error (RMSE)', fontsize=14, fontweight='bold', pad=15)  
 axes[0].set\_ylabel('RMSE ($)', fontsize=11, fontweight='bold')  
 axes[0].tick\_params(axis='x', rotation=45, labelsize=10)  
 axes[0].tick\_params(axis='y', labelsize=10)  
 axes[0].grid(axis='y', alpha=0.3)  
   
 # Add value labels on bars  
 for i, v in enumerate(rmse\_values):  
 axes[0].text(i, v + max(rmse\_values)\*0.02, f'${v:,.0f}', ha='center', va='bottom', fontsize=10, fontweight='bold')  
   
 # R² Comparison  
 axes[1].bar(model\_names, r2\_values, color=colors[:len(model\_names)], alpha=0.7, edgecolor='black')  
 axes[1].set\_title('R² Score (Coefficient of Determination)', fontsize=14, fontweight='bold', pad=15)  
 axes[1].set\_ylabel('R² Score', fontsize=11, fontweight='bold')  
 axes[1].tick\_params(axis='x', rotation=45, labelsize=10)  
 axes[1].tick\_params(axis='y', labelsize=10)  
 axes[1].set\_ylim([0, 1])  
 axes[1].grid(axis='y', alpha=0.3)  
 axes[1].axhline(y=0.5, color='red', linestyle='--', alpha=0.5, linewidth=2, label='Baseline (0.5)')  
 axes[1].legend(fontsize=10)  
   
 # Add value labels  
 for i, v in enumerate(r2\_values):  
 axes[1].text(i, v + 0.02, f'{v:.4f}', ha='center', va='bottom', fontsize=10, fontweight='bold')  
   
 # MAE Comparison  
 axes[2].bar(model\_names, mae\_values, color=colors[:len(model\_names)], alpha=0.7, edgecolor='black')  
 axes[2].set\_title('Mean Absolute Error (MAE)', fontsize=14, fontweight='bold', pad=15)  
 axes[2].set\_ylabel('MAE ($)', fontsize=11, fontweight='bold')  
 axes[2].tick\_params(axis='x', rotation=45, labelsize=10)  
 axes[2].tick\_params(axis='y', labelsize=10)  
 axes[2].grid(axis='y', alpha=0.3)  
   
 # Add value labels  
 for i, v in enumerate(mae\_values):  
 axes[2].text(i, v + max(mae\_values)\*0.02, f'${v:,.0f}', ha='center', va='bottom', fontsize=10, fontweight='bold')  
   
 plt.tight\_layout()  
   
 # Save in multiple formats  
 fig.savefig('figures/model\_performance\_comparison.png', dpi=300, bbox\_inches='tight')  
 fig.savefig('figures/model\_performance\_comparison.svg', format='svg', bbox\_inches='tight')  
 fig.savefig('figures/model\_performance\_comparison.html', format='svg', bbox\_inches='tight')  
   
 plt.show()  
   
 display(Markdown(f"""  
::: {{.callout-tip}}  
## 📊 Model Performance Visualization Complete  
  
Compared \*\*{len(successful\_models)} models\*\* across three key metrics (RMSE, R², MAE)  
  
### Figures saved to:  
 - `figures/model\_performance\_comparison.png`  
 - `figures/model\_performance\_comparison.svg`  
 - `figures/model\_performance\_comparison.html`  
:::  
"""))  
else:  
 display(Markdown("""  
::: {.callout-warning}  
## ⚠️ No Models Available for Visualization  
No successful models to visualize. Please check model training results.  
:::  
"""))



|  |
| --- |
| 📊 Model Performance Visualization Complete |
| Compared **3 models** across three key metrics (RMSE, R², MAE) 4.1.1 Figures saved to:  * figures/model\_performance\_comparison.png * figures/model\_performance\_comparison.svg * figures/model\_performance\_comparison.html |

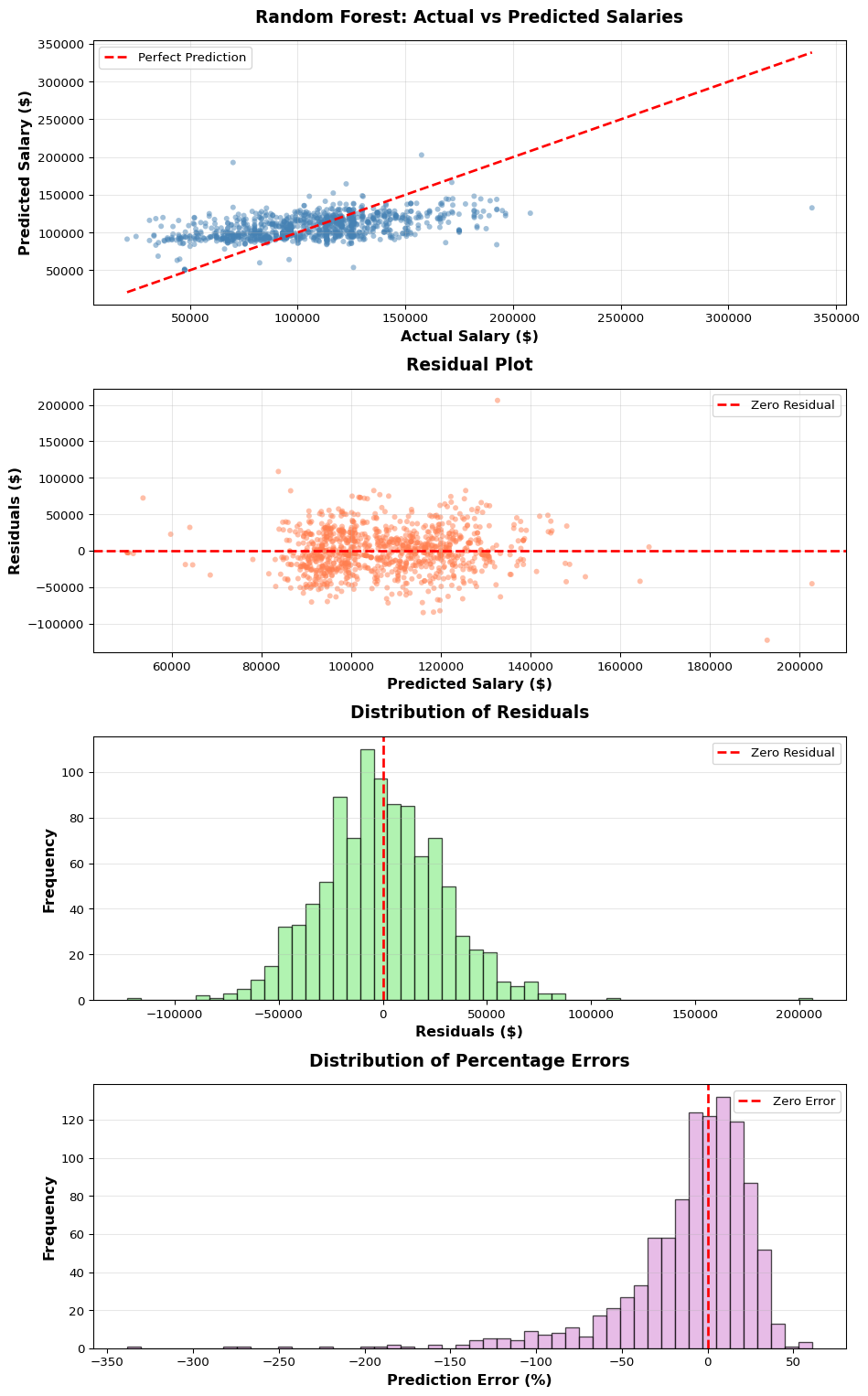
## 4.2 Feature Importance Visualization

# Visualize feature importance from Random Forest  
if importance\_df is not None:  
 # Create a copy for display with cleaned feature names  
 display\_df = importance\_df.copy()  
   
 # Clean feature names for better display  
 def clean\_feature\_name(name):  
 """Clean and shorten feature names for better visualization"""  
 # Remove list brackets and quotes  
 name = str(name).strip('[]"\' ')  
   
 # If feature name contains '=', split and clean the value part  
 if '=' in name:  
 parts = name.split('=', 1)  
 feature\_base = parts[0]  
 value = parts[1].strip('[]"\' ')  
   
 # Shorten base feature name  
 if feature\_base == 'EMPLOYMENT\_TYPE\_NAME':  
 feature\_base = 'Employment'  
 elif feature\_base == 'EDUCATION\_LEVELS\_NAME':  
 feature\_base = 'Education'  
 elif feature\_base == 'REMOTE\_TYPE\_NAME':  
 feature\_base = 'Remote'  
 elif feature\_base == 'STATE\_NAME':  
 feature\_base = 'State'  
   
 # Truncate long values  
 if len(value) > 25:  
 value = value[:22] + "..."  
   
 return f"{feature\_base}: {value}"  
 else:  
 # For numerical features, shorten the name  
 name = name.replace('MIN\_YEARS\_EXPERIENCE', 'Min Exp (yrs)')  
 name = name.replace('MAX\_YEARS\_EXPERIENCE', 'Max Exp (yrs)')  
 name = name.replace('EXPERIENCE\_MID\_YEARS', 'Mid Exp (yrs)')  
 name = name.replace('POSTING\_DURATION', 'Post Duration (days)')  
 name = name.replace('SALARY\_POSTING\_AVAILABLE', 'Salary Posted')  
 name = name.replace('DURATION', 'Duration')  
 return name  
   
 display\_df['Feature\_Display'] = display\_df['Feature'].apply(clean\_feature\_name)  
   
 # Create vertical layout (2 rows, 1 column) for better DOCX compatibility  
 fig, axes = plt.subplots(2, 1, figsize=(10, 12))  
   
 # Top 15 features bar chart  
 top\_15 = display\_df.head(15)  
 axes[0].barh(range(len(top\_15)), top\_15['Importance'], color='steelblue', alpha=0.7, edgecolor='black')  
 axes[0].set\_yticks(range(len(top\_15)))  
 axes[0].set\_yticklabels(top\_15['Feature\_Display'], fontsize=9)  
 axes[0].invert\_yaxis()  
 axes[0].set\_xlabel('Importance Score', fontsize=12, fontweight='bold')  
 axes[0].set\_title('Top 15 Most Important Features', fontsize=14, fontweight='bold', pad=15)  
 axes[0].grid(axis='x', alpha=0.3)  
   
 # Add value labels  
 for i, v in enumerate(top\_15['Importance']):  
 axes[0].text(v + 0.001, i, f'{v:.4f}', va='center', fontsize=9, fontweight='bold')  
   
 # Cumulative importance  
 cumulative\_importance = importance\_df['Importance'].cumsum()  
 axes[1].plot(range(1, len(cumulative\_importance) + 1), cumulative\_importance,   
 marker='o', markersize=4, linewidth=2, color='darkgreen')  
 axes[1].axhline(y=0.8, color='red', linestyle='--', alpha=0.7, linewidth=2, label='80% Threshold')  
 axes[1].axhline(y=0.9, color='orange', linestyle='--', alpha=0.7, linewidth=2, label='90% Threshold')  
 axes[1].set\_xlabel('Number of Features', fontsize=12, fontweight='bold')  
 axes[1].set\_ylabel('Cumulative Importance', fontsize=12, fontweight='bold')  
 axes[1].set\_title('Cumulative Feature Importance', fontsize=14, fontweight='bold', pad=15)  
 axes[1].grid(alpha=0.3)  
 axes[1].legend(fontsize=10)  
   
 # Find number of features for 80% and 90%  
 features\_80 = (cumulative\_importance >= 0.8).idxmax() + 1  
 features\_90 = (cumulative\_importance >= 0.9).idxmax() + 1  
   
 axes[1].annotate(f'{features\_80} features\n(80%)',   
 xy=(features\_80, 0.8), xytext=(features\_80 + 10, 0.75),  
 arrowprops=dict(arrowstyle='->', color='red', lw=1.5),  
 fontsize=10, ha='left')  
   
 plt.tight\_layout()  
   
 # Save in multiple formats  
 fig.savefig('figures/feature\_importance.png', dpi=300, bbox\_inches='tight')  
 fig.savefig('figures/feature\_importance.svg', format='svg', bbox\_inches='tight')  
 fig.savefig('figures/feature\_importance.html', format='svg', bbox\_inches='tight')  
   
 plt.show()  
   
 display(Markdown(f"""  
::: {{.callout-tip}}  
## 📊 Feature Importance Insights  
  
- \*\*Top features\*\* drive prediction accuracy  
- \*\*{features\_80} features\*\* explain 80% of importance  
- \*\*{features\_90} features\*\* explain 90% of importance  
  
### Figures saved to:  
 - `figures/feature\_importance.png`  
 - `figures/feature\_importance.svg`  
 - `figures/feature\_importance.html`  
:::  
"""))  
else:  
 display(Markdown("""  
::: {.callout-warning}  
## ⚠️ Feature Importance Data Not Available  
Feature importance visualization requires Random Forest model with successful training.  
:::  
"""))



|  |
| --- |
| 📊 Feature Importance Insights |
| * **Top features** drive prediction accuracy * **55 features** explain 80% of importance * **54 features** explain 90% of importance  4.2.1 Figures saved to:  * figures/feature\_importance.png * figures/feature\_importance.svg * figures/feature\_importance.html |

## 4.3 Prediction Analysis



|  |
| --- |
| 📊 Prediction Quality Metrics |
| * **Mean Residual:** $-430.04 * **Std Dev of Residuals:** $29,387.83 * **Mean Absolute % Error:** 26.10% * **Predictions within ±10%:** 31.4% * **Predictions within ±20%:** 55.8%   Figures saved to:   * figures/prediction\_analysis.png * figures/prediction\_analysis.svg * figures/prediction\_analysis.html |

## 4.4 Summary and Insights

|  |
| --- |
| 🎯 Key Findings and Insights |
| 4.4.1 Model Performance  * **Best Performing Model:** Random Forest * **R² Score:** 0.2384 (explains 23.8% of salary variance) * **RMSE:** $29,376.54 * **Number of Models Evaluated:** 3  4.4.2 Feature Insights **Most Important Features:**   1. **MIN\_YEARS\_EXPERIENCE:** 0.2013 2. **MAX\_YEARS\_EXPERIENCE:** 0.1177 3. **DURATION:** 0.1028   **Total Features Analyzed:** 93 4.4.3 Data Quality  * **Training Records:** 4,021 * **Testing Records:** 1,018 * **Total Records Processed:** 5,039  4.4.4 Recommendations  * The Random Forest model shows **weak** predictive power for salary estimation * Experience-related features appear to be key drivers of salary predictions * Geographic location (STATE\_NAME) and employment type significantly influence salary ranges * Consider collecting additional features like company size, industry sector, and specific skills for improved predictions |